UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

INTERFACE CORBA COM MIDDLEWARE BASIC SYSTEM

Sara Cristina Lopes do Patrocínio Silva

VERSÃO PÚBLICA

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computador

2011

UNIVERSIDADE DE LISBOA

Faculdade de Ciências
Departamento de Informática

INTERFACE CORBA COM MIDDLEWARE BASIC SYSTEM

Sara Cristina Lopes do Patrocínio Silva

PROJECTO

Trabalho orientado pelo Prof. Doutor Mário João Barata Calha
e co-orientado por Eng. José dos Santos Mestre Vermelhudo

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computador

2011

Resumo

Com o passar dos anos tem aumentado o grau de heterogeneidade de software nos sistemas distribuídos. De forma a minimizar os problemas que podem advir deste aumento, utiliza-se normalmente um *middleware* que separa a camada aplicacional da componente sistema operativo garantindo assim a independência e interoperabilidade entre os vários tipos de nós de um sistema distribuído.

O sistema de controlo de tráfego aéreo da NAV Portugal E.P.E. LISATM é constituído por um grande número de componentes existindo uma grande heterogeneidade entre eles. Este sistema LISATM tem como principal função o de apresentar informação que permita efectuar o controlo de tráfego aéreo, sendo necessário que todas as componentes troquem informação entre si tendo para isso que existir interoperabilidade entre elas. Uma forma de garantir a interoperabilidade desejada foi recorrendo a um *middleware* proprietário, o *Basic System.*

Ao longo do tempo têm surgido novas tecnologias e novos sistemas operativos, tornando-se necessário garantir que o *Basic System* se mantém compatível com o novo software e hardware que surge, de modo a continuar desempenhar a função para a qual foi desenvolvida.

O CORBA é um *middleware* desenvolvido segundo uma especificação da *Object Management Group*. Existem várias implementações desta especificação que também tem vindo a ser actualizada. O objectivo deste trabalho é a criação uma interface CORBA para o *middleware Basic System* de modo a mostrar que é possível utilizar o *Basic System* recorrendo a uma tecnologia conhecida, esta interface será criada sobre uam ferramenta do *middleware*, responsável por gerir as bases de dados controladas pelo *Basic System*.

Com a realização deste projecto foi possível operar sobre as bases de dados controladas pelo *Basic System* recorrendo à interface CORBA desenvolvida. Ficou provado que é possível adaptar o *middleware* proprietário para que se possa responder ao avanço da tecnologia, dando assim continuidade a um *legacy system*.

Palavras-chave: *Middleware*, *Basic System*, CORBA, interoperabilidade

Abstract

With the years passing the degree of heterogeneity in distributed systems has increased. In order to deal with this problem a middleware layer is used to separate the application layer from the operating system ensuring application independence and interoperability between the various nodes of a distributed system.

The air traffic control system at NAV Portugal E.P.E., LISATM, consists in a large number of nodes and there is a great heterogeneity between them. The system's main function is to present information that allows the air traffic controller to take quality decisions (i.e. to separate traffic in a safe manner) based on the presented traffic and flight plan information. This requires that all nodes exchange information with each other in a timely manner. A way to guarantee the desired interoperability is using the NAV proprietary middleware Basic System.

Over the last few years new technologies and operating systems have been introduced to the LISATM system given rise to changes in the middleware to comply with new requirements imposed by the industry

The CORBA Middleware is a specification from the Object Management Group. There are several implementations of this specification that keeps evolving. The aim of this work is to demonstrate that it is viable to incorporate a CORBA interface, as a pluggable task, into the middleware Basic System. The task chosen to which this interface shall be added is the one responsible for managing databases inside a program context on top of the middleware layer Basic System.

With the completion of this project it was possible to perform operations on the databases controlled by the middleware using the CORBA interface. This project has proved that it is possible to adapt the proprietary middleware and incorporate recent technology without repercussion to the existing legacy system.

Keywords: Middleware, Basic System, CORBA, interoperability

Conteúdo

Lista de Acrónimos

SIGLA	SIGNIFICADO
ATM	*Air Traffic Management*
BS	*Basic System*
BSS	*Basic System Shell*
CORBA	*Common Object Request Broker Architecture*
DDC	Data Description Compiler
DDS	*Data Distribution Service*
DSM	*Data Store Manager*
DSTI	Direcção de Sistemas e Tecnologias de Informação
FD	*Functional Destination*
IDL	*Interface Description Language*
IOR	*Interoperable Object Reference*
LISATM	*Lisbon Air Traffic Management System*
NAV	Empresa prestador de serviço de Navegação Aérea em Portugal
OMG	*Object Management Group*
ORB	*Object Request Broker*
PM	*Program Manager*
RPC	*Remote Procedure Call*
SISINT	Sistemas – Interface com o utilizador
SISLOG	Sistemas – Logística
SISPRO	Sistemas – Produção de Software
SISQUA	Sistemas – Qualidade e Safety

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este documento corresponde ao Relatório Final desenvolvido para o Projecto de Engenharia Informática do Mestrado em Engenharia Informática da Faculdade de Ciências da Universidade de Lisboa e pretende apresentar o trabalho desenvolvido na NAV Portugal E.P.E..

1.1 Motivação

Um dos problemas fundamentais na área de sistemas distribuídos é como lidar com o alto grau de heterogeneidade entre os diversos nós. Cada nó pode ter hardware, sistema operativo, rede, administração e políticas de segurança diferentes e é necessário garantir a interoperabilidade entre estes nós. Esta interoperabilidade é conseguida através do uso de um *middleware*. O *middleware* separa a camada aplicacional da camada sistema operativo melhorando a portabilidade de cada aplicação, sendo apenas o *middleware* o único com necessidade de ser portado. Sendo assim torna-se possível a interacção entre os múltiplos processos.

O LISATM [9](*Lisbon Air Traffic Management System*) é o sistema utilizado no centro de controlo de tráfego aéreo de Lisboa como suporte ao controlo de tráfego áereo utilizado pelos serviços de tráfego aéreo. Este sistema é responsável por fornecer ao controlador funções de auxílio no controlo de tráfego aéreo tal como a posição das aeronaves, a sua identificação, velocidade, altitude e outras características do plano de voo. Além da informação referida também é possível ao controlador visualizar informação sobre a meteorologia e dados de coordenação com o aeroporto. De modo a garantir todos estes serviços o LISATM é composto por um conjunto de entidades internas e externas com as quais interage de modo a obter e fornecer dados importantes para os serviços de sistema de tráfego aéreo. Entende-se por entidades internas aquelas

que são da responsabilidade da NAV e externas como aquelas que são da responsabilidade de outras empresas (externas à NAV).

O *Basic System* (secção 3.2) é o *middleware* proprietário utilizado na NAV Portugal EPE, que permite a comunicação entre os vários componentes do LISATM. Também é oferecido pelo BS serviços como *Share Memory*, Supervisão, *File Management* entre outros.

No entanto sendo o *Basic System* um *Legacy System* e o LISATM um sistema baseado em *Hardware* e *Software* disponível no mercado e susceptível de actualização ao ritmo da evolução tecnológica é necessário dar continuidade a este sistema enquanto se acompanha a evolução da tecnologia. É necessário investir na definição de interfaces padronizadas de modo a permitir albergar componentes de diferentes fabricantes. Para isso é necessário descrever as interfaces de um objecto numa linguagem neutra.

Tendo em conta este cenário e considerando a regulamentação em vigor a utilização do CORBA é uma opção que cumpre os requisitos de interoperabilidade. O CORBA é uma norma definida pelo *Object Management Group* que permite a comunicação entre vários componentes de *software* escritos em linguagens de programação diferentes.

1.2 Objectivos

Este projecto tem como principal objectivo criar uma interface CORBA com o *Middleware Basic System*. Os requisitos do cliente referem que esta interface seja uma interface CORBA e que o bloco estendido seja o serviço de *Data Store Manager*. O objectivo principal é criar uma prova de conceito de modo a verificar e demonstrar o funcionamento destas duas camadas.

Para atingir este objectivo é necessário numa primeira fase realizar o estudo do *Basic System* de modo a conhecer os seus componentes, o seu funcionamento e o modo de interacção entre eles. Também é necessário estudar o subcomponente *Data Store Manager* bem como as ferramentas a este associado.

Depois deste estudo, é necessária uma análise do problema de modo a poder definir as interfaces necessárias. O objectivo desta fase é a identificação do problema encontrado, de modo a perceber as alterações/adaptações que serão necessárias efectuar. Para isto é necessário identificar os actores do sistema, bem como identificar requisitos e construir alguns casos de uso.

O passo seguinte será é a fase do desenho do sistema a desenvolver garantindo que esta arquitectura respeita os requisitos identificados.

Após esta definição é necessário produzir então um protótipo de modo a simular o subsistema. Para isso será necessário proceder à escolha de um ORB que corresponda às necessidades da empresa. Posto isto, serão desenvolvidos adaptadores em C++ e Java para obter dados do Sistema e posteriormente preparar a plataforma com comunicações CORBA, confirmando-se o adequado funcionamento do sistema.

Caso esta prova de conceito tenha um resultado positivo será necessário realizar testes de desempenho de modo a avaliar a viabilidade da solução proposta.

1.3 Planeamento

O planeamento inicial para a realização deste projecto era o seguinte:

ID	Descrição	Duração (dias)	Fase
E1	Formação no sistema LISATM	1	
E2	Introdução ao Middleware Basic System da NAV Portugal	4	
E3	Introdução ao Sistema de gestão de Qualidade	1	
E4	Preparar ambiente de trabalho e ferramentas de desenvolvimentio	4	
E5	Baseado nas especificações CORBA do ICOG criar um protótipo para simular um subsistema do LISATM	53	
E6	Desenvolver adaptadores em Java e C++ para obter dados do sistema	30	
E7	Preparar plataforma com comunicações CORBA que verifique o funcionamento para multiplos clientes	35	
E8	Preparar documentação e especificação de software e de interface	30	
E9	Produção do relatório preliminar	5	
E10	Produção do realatório final em versão confidencial	15	
E11	Produção do relatório final em versão pública	2	

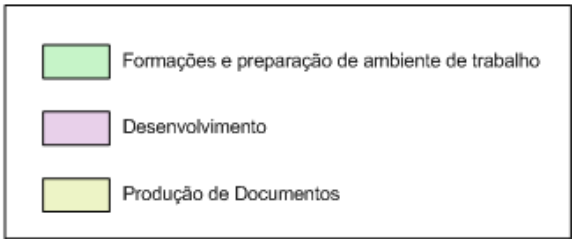

Tabela 1 – Planeamento Inicial

Este planeamento é composto por três fases. Uma fase inicial de formações e preparação de ambiente de trabalho (E1-E4), uma de desenvolvimento do projecto em si (E5-E7) e uma de produção de relatórios (E8-E11).

Nesta primeira fase foram introduzidos conceitos relacionados com o negócio em que a NAV se insere o controlo de tráfego aéreo. Também foram apresentadas algumas metodologias de trabalho utilizadas na NAV. Além das formações inicialmente previstas também foram leccionadas formações sobre *Data Link* e na *framework* ATC. O tempo previsto para a formação de introdução ao *middleware Basic System* também foi estendido devido a tratar-se de uma formação teórico-prática. Posto isto, o tempo previsto inicialmente para formações foi estendido.

Na segunda fase, foi realizada a análise do problema, estudo das opções e tomadas decisões de desenho. Também foi implementado o protótipo, de modo a criar uma prova de conceito para avaliar a compatibilidade entre o *Basic System* e o CORBA bem como testado o seu desempenho. Nesta fase foram implementados vários protótipos, desde protótipos com dados fictícios a protótipos com dados reais.

No entanto, a tarefa E5 teve de ser suprimida visto a NAV Portugal E.P.E. não ter sido possível a licença para utilizar as especificações do ICOG. Os direitos sobre esta especificação foram adquiridos pela indústria e a NAV Portugal E.P.E. não tem uma participação directa no projecto, não sendo possível então a realização esta tarefa.

Por fim, a última fase foi uma fase de produção de documentação e relatórios finais de avaliação. Toda a informação foi documentada numa plataforma online interna na NAV.

1.4 Instituição de Acolhimento

A NAV Portugal, EPE [10] é a empresa responsável pela prestação de serviços de tráfego aéreo nas RIV (Região de Informação de Vôo) de Lisboa e Santa Maria, sob a responsabilidade Portuguesa. É também da sua responsabilidade garantir o cumprimento da regulamentação nacional e internacional nas melhores condições de segurança, optimizando capacidades, privilegiando a sua eficiência sem descurar as preocupações ambientais.

Foram identificados como grandes objectivos da empresa os seguintes: optimização dos padrões de segurança do tráfego aéreo, resposta adequada à procura do

tráfego, melhoria da relação custo/eficácia, evolução para a excelência, no âmbito da gestão e qualidade e afirmação da influência de Portugal no Atlântico Norte.

A empresa está sediada em Lisboa junto ao Aeroporto Internacional de Lisboa, onde se encontra também o seu Centro de Controlo de Tráfego Aéreo e o seu Centro de Formação. Na região Autónoma dos Açores, mais concretamente na Ilha de Santa Maria, está situado o Centro de Controlo Oceânico. A NAV possui ainda outras infra-estruturas com Serviços de Tráfego Aéreo a funcionar nas Torres de Controlo dos Aeroportos de Lisboa, Porto, Faro, Funchal, Porto Santo, Santa Maria, Ponta Delgada, Horta, Flores e no Aeródromo de Cascais. Para além disso também possui um vasto conjunto de equipamentos e instalações técnicas em vários pontos de Portugal Continental e Regiões Autónomas.

Este projecto integra-se na DSTI – Direcção de Sistemas e Tecnologias de Informação, patrocinadora do projecto, tem a sua principal missão no desenvolvimento e manutenção de software dos sistemas ATM (*Air Traffic Management*) e é composta pelos seguintes serviços:

- SISPRO – responsável pela produção de *software*
- SISINT – responsável pela captura de requisitos do utilizador
- SISLOG – responsável pela área de logística
- SISQUA – responsável pela gestão de qualidade e *safety*.

Este projecto integra-se no âmbito das responsabilidades do serviço SISPRO do qual o seu responsável é o co-orientador deste projecto de estágio.

1.5 Organização do Documento

Este documento apresenta a seguinte organização:

Capítulo 2 – Este capítulo irá abordar algumas metodologias de trabalho da direcção. Desde o processo de desenvolvimento de *software* ao processo de qualidade.

Capítulo 3 – Neste capítulo irá ser abordado algum do trabalho já existente permitindo ao leitor integrar-se no tema. Será detalhado os sistemas já existentes na NAV bem como as tecnologias a aplicar.

Capítulo 4 – Neste capítulo será apresentado o trabalho realizado ao longo deste projecto.

Capítulo 5 – Este capítulo apresenta as principais contribuições que este projecto trouxe bem como os resultados obtidos e a perspectiva de evolução futura.

Capítulo 2

Metodologias

Este capítulo pretende apresentar algumas das metodologias utilizadas na NAV Portugal E.P.E.. Assim será abordado o modelo de desenvolvimento de software em V e o processo de qualidade existente.

2.1 Modelo de Desenvolvimento de Software

Em muitos dos projectos realizados na DSTI, o modelo de desenvolvimento de *software* utilizado é o Modelo em V [5]. Este modelo é considerado uma extensão do modelo em cascata [1], mas assume que a fase de testes é executada durante todo o processo de desenvolvimento, permitindo assim melhorar a qualidade do produto.

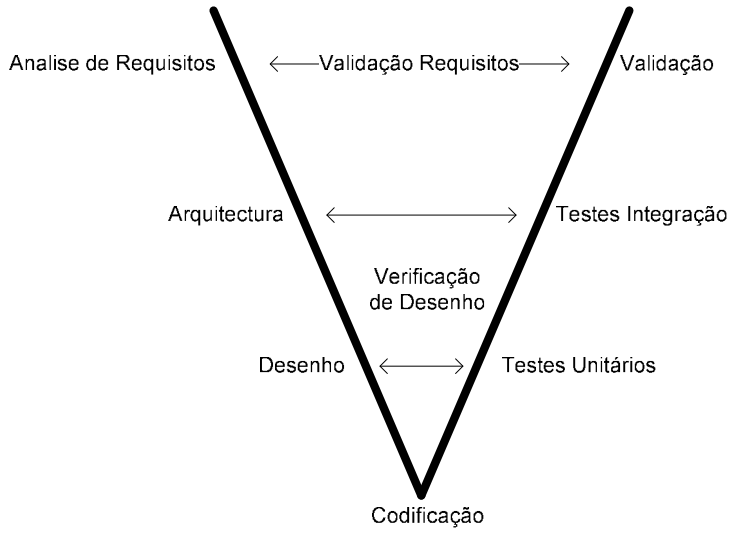

Figura 1 – Modelo em V

A Figura 1 mostra-nos as várias fases que este processo possui:

- Análise de Requisitos/Especificações: Nesta fase é feito o levantamento dos requisitos funcionais e não funcionais do sistema. Para isso é necessário analisar as necessidades do cliente validando simultaneamente esses requisitos.
- Arquitectura: Esta é a fase de desenho da arquitectura do Sistema. Nesta fase é necessário avaliar a sua integração no sistema global e como garantir o teste desta integração.
- Desenho: Baseado na arquitectura de sistema definida anteriormente, é necessário detalhar cada módulo do *software* a desenvolver. Em simultâneo com esta fase é necessário planear como testar estas funcionalidades individualmente.
- Codificação: Com base no desenho codifica-se então cada funcionalidade estabelecida anteriormente.
- Testes Unitários: Esta é a fase onde se testa individualmente e cada função codificada de modo a verificar se corresponde ao especificado.
- Testes de Integração: Durante esta fase são testadas as funções implementadas nos diversos subsistemas em conjunto e quando integradas no sistema.
- Validação: Nesta última fase são verificados e validados os requisitos iniciais do cliente

Com este modelo estas três últimas fases de teste e validação tornam-se bastante mais fáceis pois são planeadas simultaneamente com as especificações respectivas.

A utilização deste modelo [2] tem com objectivo minimizar os riscos e custos do projecto, melhorando e garantido a sua qualidade. Também é um objectivo da utilização deste modelo de desenvolvimento de software aumentar a comunicação entre todos os intervenientes do projecto. Estes objectivos são atingidos devido ao controlo que se tem sobre o projecto bem como devido à transparência que o modelo oferece.

A utilização do modelo em V na DSTI deve-se à complexidade dos projectos existentes. Os artefactos resultantes das fases pré-codificação (Análise de requisitos, arquitectura e desenho) tornam a tarefa de codificar bastante mais simples pois o projecto encontra-se bastante bem definido.

No Capítulo 4 é possível fazer uma ligação com este modelo, pois a ordem como os resultados são apresentados tenta seguir a ordem das fases do modelo em V.

2.2 Processo de Qualidade

O POP (Procedimento OPeracional) 20 – Prestação de Serviços de Desenvolvimento de Sistemas [11] – é o processo de qualidade utilizado na DSTI no âmbito do SGQE (Sistema de Gestão da Qualidade e Ambiente) na qual a NAV está certificada. Este processo serve para uniformizar os procedimentos de desenvolvimento de sistema

2.3 Sumário

Este capítulo descreveu o modelo de desenvolvimento de software seleccionado pela NAV Portugal E.P.E. para este projecto bem como descreveu o processo de qualidade utilizado na DSTI.

O próximo capítulo irá apresentar o trabalho já existente e relacionado com o projecto em questão.

Capítulo 3

Trabalho relacionado

O objectivo deste capítulo é descrever sistemas e especificações relacionadas com o projecto. Sendo assim será apresentado o *middleware* proprietário da NAV Portugal E.P.E., o BS (*Basic System*), a especificações CORBA e *Data-Distribution Service* do OMG.

3.1 LISATM

O LISATM (*Lisbon Air Traffic Management System*) é o sistema principal existente no centro de controlo de tráfego aéreo de Lisboa. Versões deste sistema também se encontram presentes nas diversas torres de controlo do continente e da Madeira, bem como em Santa Maria nos Açores, sendo estas duas últimas denominadas TWRATM (*Tower Air Traffic Management System*) e ATLATM (*Atlantic Air Trafic Management System*).

Sendo o LISATM um dos sistemas principais no que diz respeito ao controlo de tráfego aéreo da FIR (*Flight Information Region*) Portuguesa, é da sua responsabilidade disponibilizar uma interface ao controlador com uma vista radar do espaço aéreo nacional. Além desta vista radar o controlador tem também disponível informação sobre as aeronaves como por exemplo identificação, posição, velocidade, altitude, plano de voo, rota, entre outros dados como o estado do tempo e do aeroporto.

Tal como todo o ambiente evolvente ao controlo de tráfego aéreo o LISATM é bastante complexo sendo constituído por bastantes componentes independentes mas com o objectivo de funcionarem com um todo. A Figura 2 – Principais Componentes LISATM mostra algumas das componentes principais do LISATM estando todas ligadas entre si.

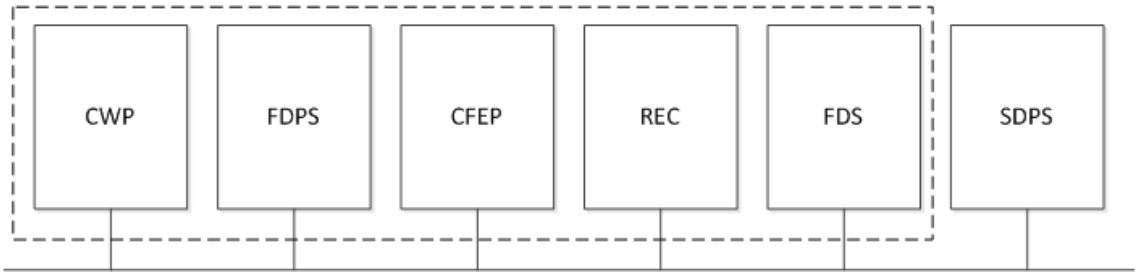

Figura 2 – Principais Componentes LISATM

De seguida é apresentada uma explicação muito resumida de cada uma das componentes:

- CWP – *Controller Working Position*: Componente responsável por apresentar a vista radar e permitir a interacção ao controlador de tráfego aéreo.
- FDPS – *Flight Data Processing System*: Servidor de planos de voo, esta componente recebe os planos de voo e distribui-os pelas restantes componentes do sistema.
- CFEP – *Communication Front End Processor*: *Front End* responsável por efectuar comunicações com entidades exteriores ao sistema.
- REC - *RECording*: Responsável por efectuar gravação do que acontece nas componentes do LISATM de forma reproduzir algum problema que aconteça.
- FDS – *Flight Data Section*: Componente onde é feita a gestão por um operador humano dos planos de voo.
- SDPS – *Surveillance Data Processing System*: Componente responsável por receber os dados de vigilância provenientes dos radares e fornecer estes dados ao interior do LISATM.

Sendo o LISATM tão complexo e constituído por componentes tão heterogéneas foi necessário adaptar o sistema para que, independentemente da heterogeneidade e do tamanho do sistema fosse possível que todas as componentes comunicassem entre si. É com este objectivo que surge o *middleware Basic System.*

A Figura 3 mostra onde se enquadra este *middleware* no sistema LISATM.

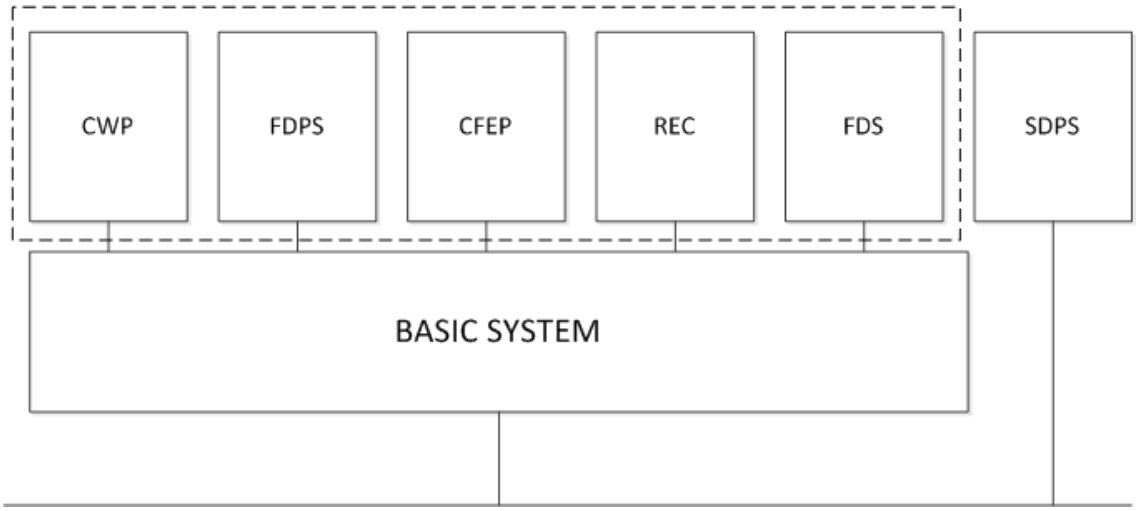

Figura 3 – Principais Componentes LISATM e *Basic System*

A secção seguinte tem o objectivo de apresentar o *B*asic *System* e de descrever o seu funcionamento.

3.2 Basic System

O BS [3] é uma plataforma *middleware* que permite a comunicação entre nós[1] heterogéneos de uma rede. O BS foi criado com o objectivo de criar normas comuns para aplicações tempo-real como é o caso das aplicações de controlo de tráfego aéreo. Isto é conseguido controlando o arranque e os recursos do sistema em cada nó, fornecendo serviços de supervisão e de troca de mensagens entre processos.

Para além disto, o BS também oferece um conjunto de produtos, sendo que alguns deles são mostrados na Figura 4**:**

[1] Entende-se por nó de um sistema como sendo um ponto de ligação aos diversos subsistemas permitindo o envio e recepção de dados

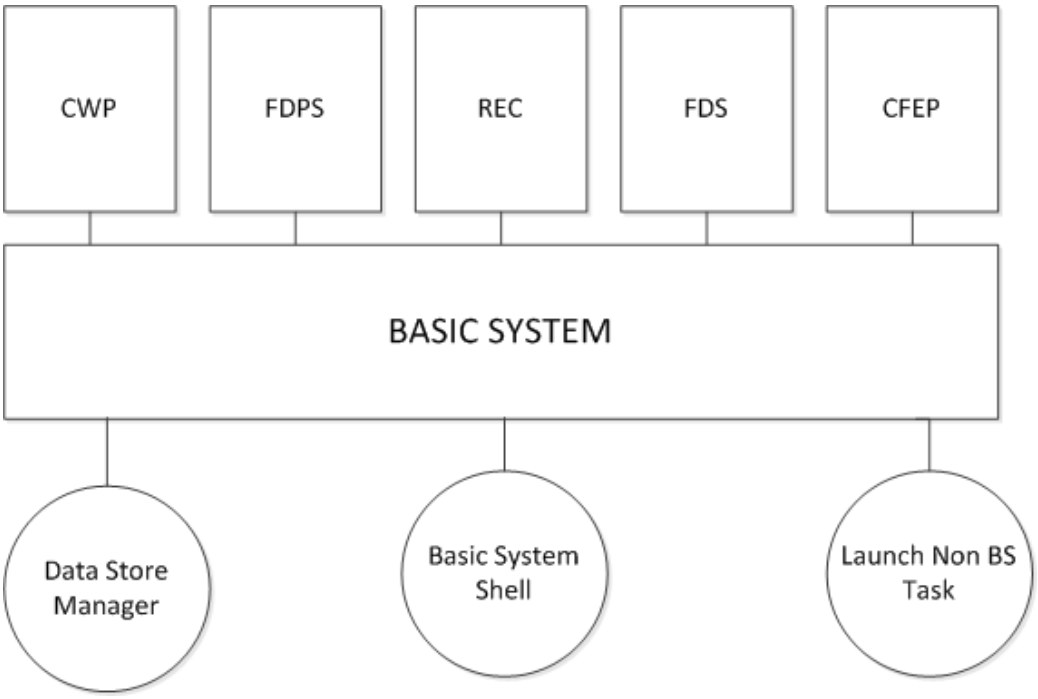

Figura 4 – Alguns produtos do *Basic System*

Os produtos apresentados na Figura 4 (*Data Store Manager*, *Basic System Shell* e *Launch Non BS Task*) são os utilizados ao longo deste projecto, sendo que não são apresentados mais pois a sua explicação tornar-se-ia bastante exaustiva e complexa.

A próxima secção detalha o produto *Data Store Manager* (DSM), pois foi sobre este que incidiu o foco deste projecto.

3.2.1 Data Store Manager

O *Data Store Manager* é um produto assente em BS, que permite a criação e manutenção de bases de dados, com garantia da integridade dos dados pelo controlo seguro das actualizações atómicas (implementa o *rollback/rollforward*). O DSM oferece um conjunto de funções que permite:

- Criar uma base de dados
- Importar dados
- Exportar dados para um ficheiro
- Alterar a estrutura da base de dados
- Redimensionar o número de objectos que a base de dados permite guardar.

O DSM utiliza o número de objectos que a base de dados permite guardar para alocar o espaço necessário no disco. Assim evita-se a corrupção devido À falta de espaço no disco.

O DSM foi o produto sobre o qual foram implementos as interfaces CORBA de modo a realizar a prova de conceito. A Figura 5 ilustra a localização da camada CORBA introduzida na realização deste projecto.

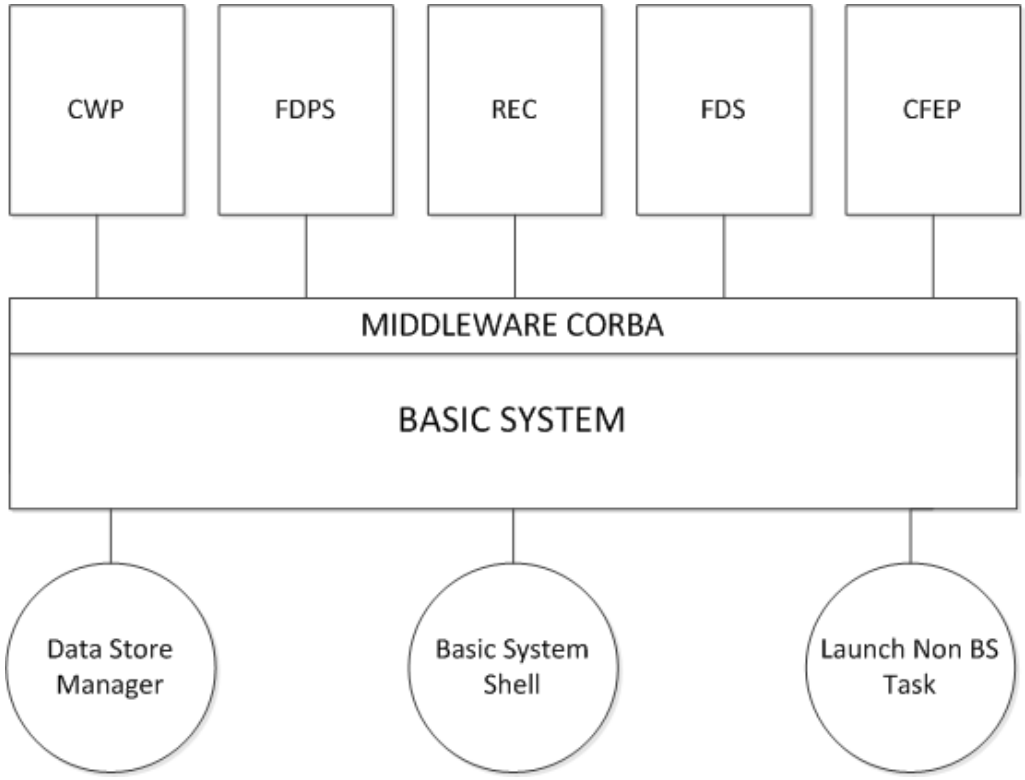

Figura 5 – Localização da Camada CORBA

3.3 CORBA

O CORBA [4,6,17] é uma arquitectura de objectos distribuídos desenvolvida pela OMG, que tem como objectivo simplificar a troca de informação entre máquinas que fazem parte de um sistema distribuído. Define um protocolo de interacção entre os clientes e os objectos remotos. Este objectivo é conseguido através da sua independência de plataforma e da linguagem de programação. Face à diversidade de *software* e *hardware* que encontramos hoje em dia, o CORBA actua de modo a que os objectos distribuídos comuniquem entre si sem a percepção de diferenças no ambiente ou contexto em que se encontram.

De uma forma simples, uma aplicação CORBA é constituída essencialmente por três componentes principais: o cliente, o ORB (*Object Request Broker*) e os objectos remotos. O ORB é a estrutura mais importante da arquitectura CORBA corresponde à camada de *middleware*, que tem por função intermediar todas as comunicações e

transferências entre o cliente e os objectos remotos. Este *middleware* possibilita que as transacções sejam transparentes para cada uma das partes envolvidas durante todo o processo de comunicação.

Para perceber melhor o conceito de *Object Request Broker* (ORB) é necessário analisar, primeiro, cada uma destas palavras isoladamente:

- *Object* – Em CORBA, um objecto é identificado de forma única através de uma referência a um objecto remoto. Todos os objectos remotos estão disponíveis aos clientes através da sua interface que descreve um conjunto de operações que podem ser invocados pelos clientes. O uso de uma interface permite ocultar ao cliente os aspectos relacionados com a implementação dos objectos. Mudanças na implementação de um objecto não implicam necessariamente, uma alteração na sua interface, e não implicam desta forma, qualquer alteração no cliente.
- *Broker* – É a entidade que actua como um mediador para outras entidades. Em CORBA, os clientes fazem pedidos aos objectos remotos. Estes pedidos são mediados e tratados pelo *broker* que se encarrega de os transportar dos clientes até aos objectos remotos e de transportar os resultados da invocação até aos clientes.
- *Request* – É um pedido efectuado por um cliente. Um cliente realiza as suas tarefas obtendo referências para objectos remotos para posteriormente invocar operações sobre estes. Os objectos referenciados podem estar na mesma maquina que o cliente ou estarem localizados noutras maquinas. A chamada de um método de um objecto remoto, para o programador, não difere da chamada de um método local. O *Basic System* (secção 3.2) também possui esta característica. Se uma tarefa comunica com outra através de um *functional destination* então é desconhecida a localização da outra tarefa. Pode estar localizado no mesmo nó como pode estar em um nó remoto e o meio de comunicação (TCP/IP, USB, RS-232, X.25, etc.) também não influencia o comportamento da tarefa.

A partir destes três conceitos, podemos verificar que o ORB actua como um mediador, que gere a interoperabilidade entre os clientes e os objectos remotos que disponibilizam serviços aos seus clientes. Por outras palavras, o ORB é uma camada de *middleware* que coordena uma infra-estrutura de objectos distribuídos.

A Figura 6 ilustra uma chamada a um método remoto oriundo de um cliente que é enviado, por intermédio do ORB, ao objecto remoto. O ORB é responsável pela localização do objecto remoto ao qual se destina o pedido feito pelo cliente. É também responsável por transportar o pedido e os parâmetros do pedido num formato aceite pelo objecto remoto e devolver ao cliente o resultado do pedido. O ORB garante a ligação entre o cliente e o servidor, independentemente da linguagem em que foram programados e do sistema operativo onde se encontram a correr.

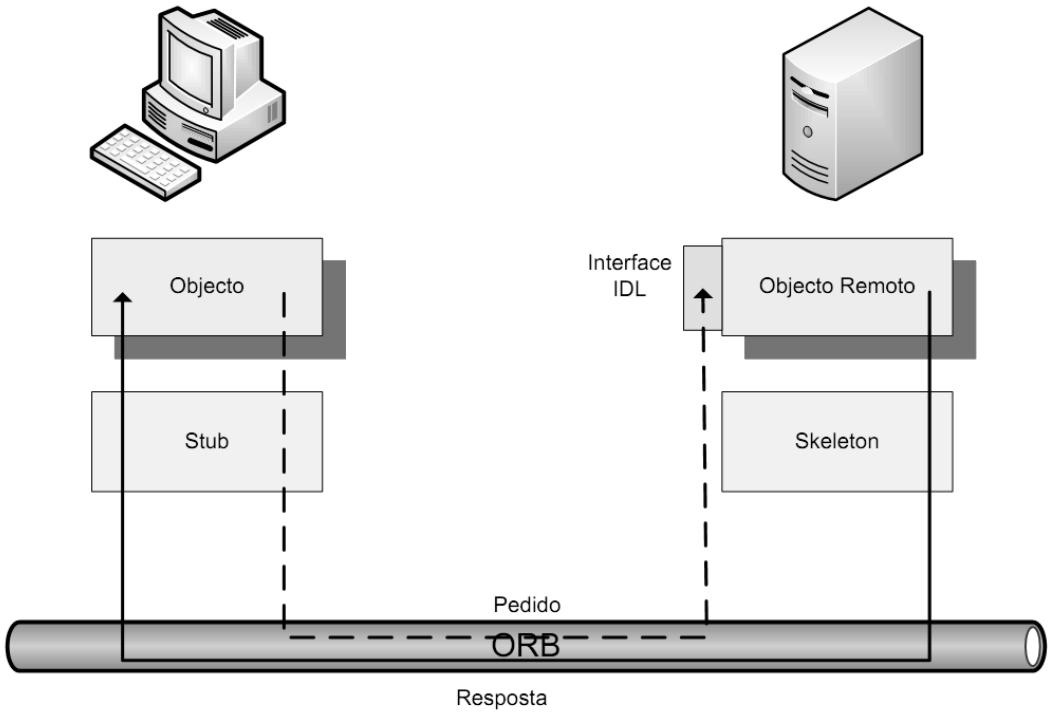

Figura 6 – Funcionamento do CORBA

De forma a permitir a interacção entre objectos distribuídos, as interfaces encontram-se descritas usando uma linguagem neutra denominada IDL. O IDL é puramente declarativo, ou seja não especifica nenhum detalhe relacionado com a implementação de um objecto. É através do IDL que se definem quais são os métodos de um objecto remoto que podem ser invocados, os seus parâmetros, valores de retorno, excepções, etc. O IDL estabelece uma espécie de contrato entre o servidor, prestador de serviços, e os seus clientes, consumidores de serviços. As interfaces descritas em IDL são utilizadas para gerar automaticamente os *stubs* e os *skeletons* através de um compilador IDL. É gerado um *stub*/*skeleton* por cada interface.

Os *stubs* são criados em tempo de compilação e permitem ao cliente realizar invocações remotas. São responsáveis por tratar dos detalhes da comunicação entre o cliente e o objecto remoto. Para o cliente o *stub* comporta-se como um *proxy* criando a ilusão de o cliente estar a usar um objecto local. É o *stub* que cria a mensagem com a identificação do método a ser invocado no objecto remoto e quais os seus parâmetros.

Os *skeletons* são a parte corresponde aos *stubs* clientes, mas do lado dos objectos remotos. O *skeleton* recebe os pedidos do ORB e invoca a implementação do método remoto pedido. Após realizar o pedido, envia uma mensagem com os resultados da invocação para o cliente.

O CORBA também oferece um conjunto de serviços disponibilizados através de interfaces, que estendem as funcionalidades básicas do ORB.

O serviço de eventos é um desses serviços e permite uma comunicação assíncrona entre objectos. Este tipo de comunicações torna-se útil em situações onde a notificação é mais importante do que a interacção. Dispõe das seguintes funcionalidades: entrega fiável, mensagens anónimas, canais de eventos e suporte a entregas do tipo *push* e *pull*. O *push* ocorre quando os eventos são "empurrados" para os consumidores pelos fornecedores. O *pull* acontece quando são os consumidores a "puxarem" os eventos dos fornecedores. Este serviço permite que objectos possam registar o seu interesse em receber eventos específicos, sendo que o *Event Channel*, fica responsável por recolher e distribuir os eventos entre os objectos do sistema.

Outro serviço importante que o CORBA oferece é o *Naming Service*. Este serviço permite a associação entre nomes definidos pelo utilizador e referências para objectos remotos, ou seja o IOR (*Interoperable Object Reference*). O IOR é um formato *standard* que permite a troca de referências entre ORB desenvolvidos por diferentes vendedores (por exemplo, Sun, IONA, Xerox, etc.).O IOR tornou possível referenciar objectos localizados em qualquer parte do mundo de uma forma não ambígua. Esta referência contém toda a informação relativa a um objecto remoto, tal como a máquina em que reside, a porta TCP/IP na qual o objecto está à escuta de pedidos e uma chave que identifica um objecto específico de entre os vários objectos que estão à escuta no mesmo porto. Para além disso também permite localizar objectos remotos com base no nome do objecto e obter uma referência para o objecto pesquisado.

Um *Naming Context* é um objecto CORBA que implementa uma entrada numa tabela que pode estar associada a uma referência para um objecto remoto ou a uma referência para outro contexto implementado pelo serviço de nomes. Isto significa que

os contextos podem formar uma hierarquia. A sequência de nós percorridos desde a raiz forma um *pathname*, um nome composto que identifica univocamente o nó alcançado. Um nó pode ter múltiplos *pathnames*.

Um cliente pode estabelecer uma ligação com o *Naming Service*, fornecer o nome de um serviço e receber uma referência para o objecto remoto associado. Nesta interacção o cliente não precisa de saber em que máquina o objecto remoto está localizado. A referência permite-lhe invocar qualquer método definido na interface do objecto remoto.

As operações básicas do *Naming Service* são o *bind* e o *resolve*. A operação *bind* é utilizada para adicionar na hierarquia o nome de um objecto e a sua referência, enquanto que a operação *resolve* permite que um cliente forneça o nome de um objecto e receba a referência desse objecto. A Figura 7 mostra estas duas operações.

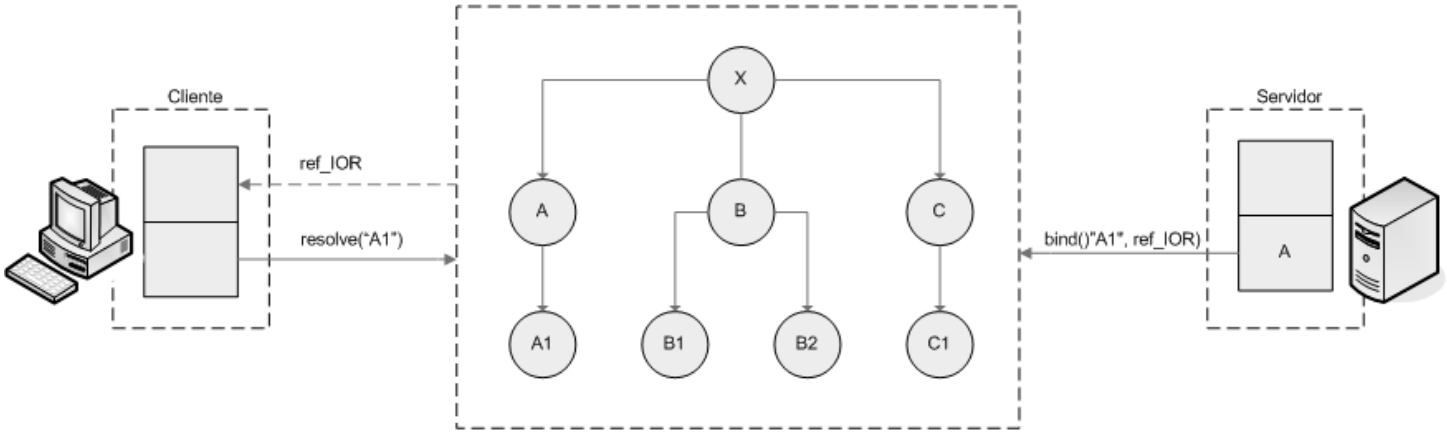

Figura 7 – Resolução de Nomes

O cliente invoca o método *resolve* passando como parâmetro o nome do objecto que procura. O *naming service* faz uma pesquisa na sua base de objectos e devolve uma referência ao cliente quando encontra o objecto que foi pedido. Um servidor, quando executado, inicia os seus objectos remotos e regista-se de imediato no *naming service* usando o método *bind*.

3.4 ORBacus

O ORBacus é um ORB que respeita as especificações CORBA definidas em "*The Common Object Request Broker: Arquitecture and Specification*" [12], "*C++ Language Mapping*" [13], "*IDL to Java Language Mapping*" e "*Portable Interceptors*" [14].

O ORBacus oferece suporte para CORBA 2.5. Também inclui algumas funcionalidades de *Quality of Service*, tais como: *Load Balancing*, *Fault Tolerance*, *Active Connection Management*, *Security*, *Concurrency* e *Dynamic Loading of Modules*. Os serviços oferecidos por este são: *Naming Service*, *Event Service* e *Property Service*.

De seguida são apresentados os vários modelos de concorrência que o ORBacus oferece.

3.4.1 Modelos de Concorrência

Um modelo de concorrência descreve como o ORB lida com a comunicação e a execução de pedidos. Existe duas categorias principais de modelos de concorrência: modelos *single-threaded* e modelos *multi-threaded*.

O ORBacus oferece diferentes modelos de concorrência a estabelecer tanto para o lado do cliente como para o lado do servidor.

Para o lado do cliente estão disponíveis dois modelos de concorrência: *Reactive* e *Threaded*. Em relação ao servidor oferece cinco modelos de concorrência: *Reactive*, *Threaded*, *Thread-per-client*, *Thread-per-request*, *Thread Pool*. Enquanto que o modelo de concorrência *Reactive* é um modelo *single-threaded* os restantes modelos são *multi-threaded*.

Reactivo

Servidores reactivos usam chamadas a operações como *select*, de modo a aceitar simultaneamente pedidos de ligação de múltiplos clientes e enviar as respostas.

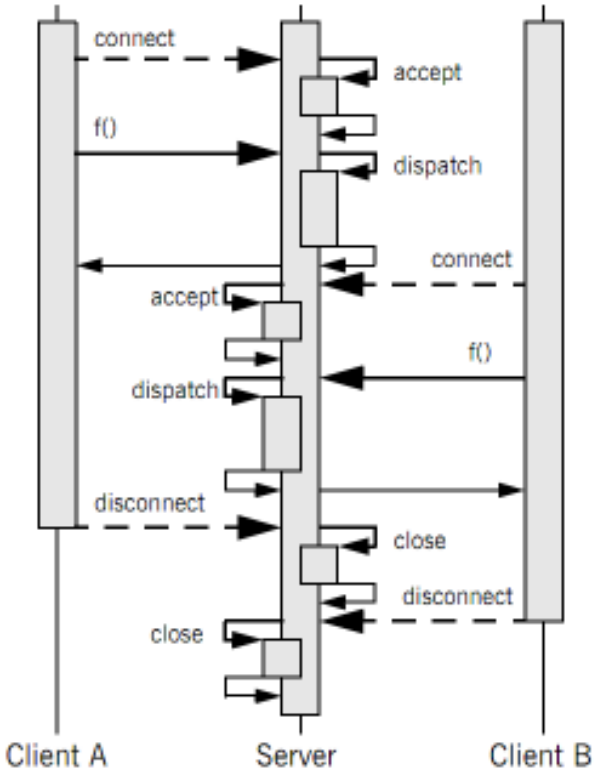

Figura 8 – Modelo de concorrência *Reactive*

Clientes reactivos também utilizam operações como o *select* para evitarem bloquear. Isto significa que quando é efectuado um pedido ou recebida uma resposta do servidor, o cliente pode enviar pedidos em *buffer* a outros servidores ou receber e colocar no *buffer* respostas. Isto é útil em operações unidireccionais.

No entanto a principal vantagem de clientes reactivos é quando é usado com um servidor reactivo em aplicações mistas cliente/servidor. Entende-se por aplicação mista cliente/servidor programas que sejam simultaneamente clientes e servidores. Sem este modelo de concorrência não é possível executar funções em simultâneo nas duas aplicações.

Este modelo é bastante rápido não existindo *overhead* tanto na criação de threads como em *context switching*.

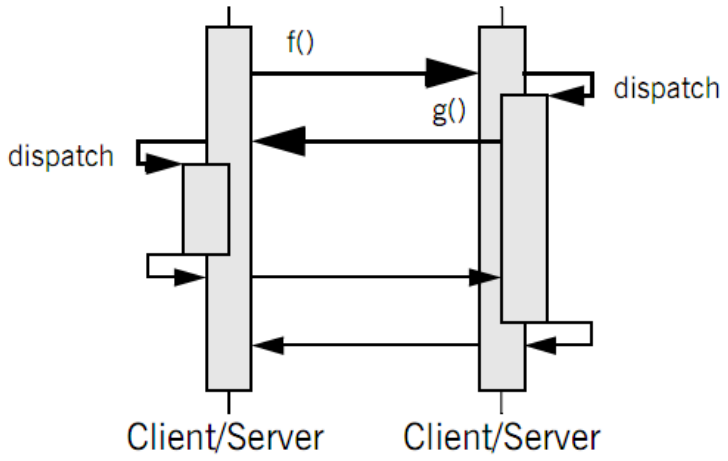

Figura 9 – Modelo de concorrência Reactive em aplicação Cliente/Servidor

Threaded

Em relação ao modelo de concorrência *Threaded* este pode ser aplicado tanto ao cliente como ao servidor. Este modelo de concorrência utiliza *threads* separadas para receber pedidos mas uma única *thread* de execução. Isto significa que pode receber pedidos simultâneos mas no entanto a execução destes pedidos são serializados. Este modelo permite que existe apenas uma *thread* com código trazendo como vantagem que o programador não tem que tratar de sincronização de *threads*. Isto significa que o código pode ser escrito num sistema *Single Threaded* mas sem perder as vantagens que o ORB tem optimizando as suas funções utilizando múltiplas *threads* internamente.

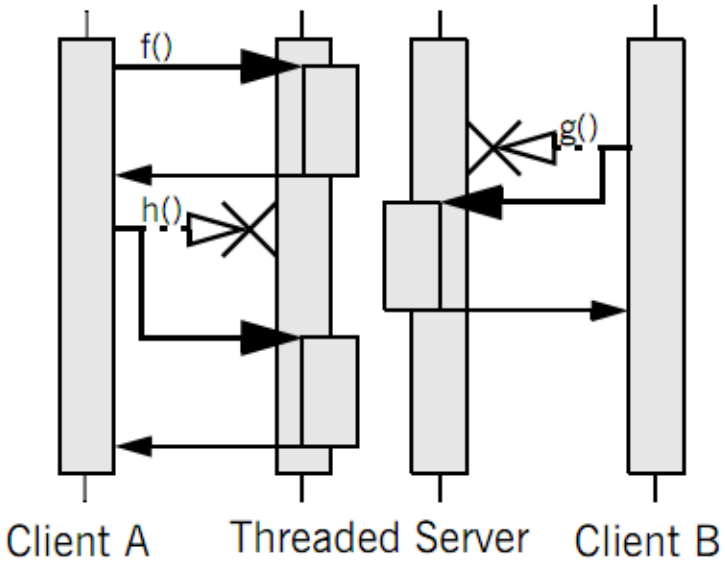

Figura 10 – Modelo de concorrência *Threaded*

Este modelo é rápido sendo que o tempo consumido na criação de threads é apenas necessário quando um novo cliente se liga e não quando é efectuado um novo pedido. No entanto a mudança de contexto consome tempo tornando este modelo mais lento que o reactivo, pelo menos num computador com apenas um processador.

Thread-per-client

Quanto ao modelo de concorrência *Thread-per-client* este está disponível para o servidor. Este modelo é muito similar ao modelo de concorrência *Threaded* do servidor, com excepção que é permitido apenas uma thread activa por cliente em código.

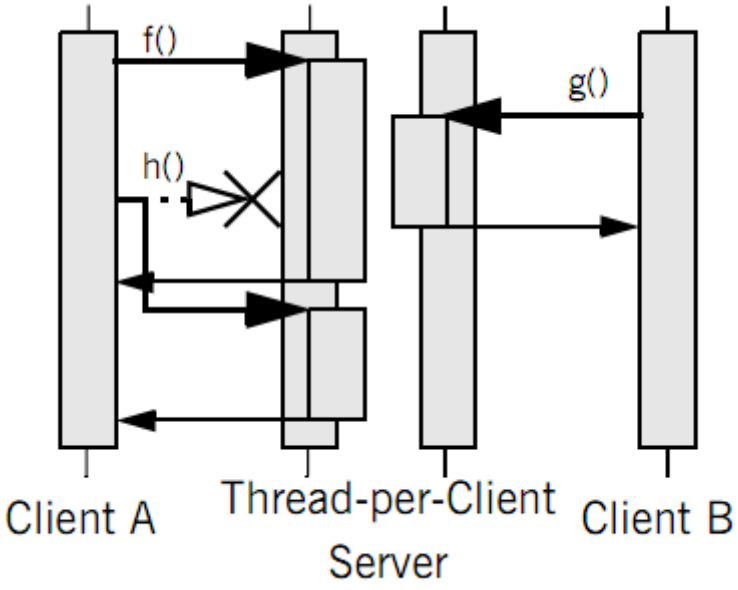

Figura 11 – Modelo de concorrência *Thread-per-Client*

Tal como o modelo *Threaded* este modelo é eficiente sendo apenas criada uma nova thread quando é aceite uma nova ligação.

Thread-per-request

No modelo de concorrência *thread-per-request* é criado uma nova thread por cada pedido. Com este modelo os pedidos nunca são atrasados, pois sempre que chega um novo é criada uma nova *thread* e o pedido é executado. Quando é enviada a resposta a *thread* é destruída.

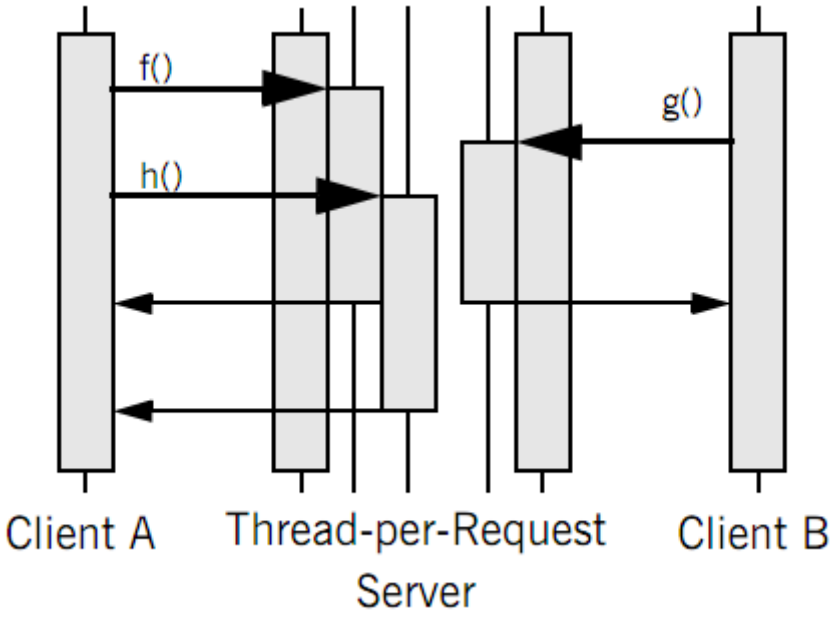

Figura 12 – Modelo de concorrência *Thread-per-Request*

Este modelo é pouco eficiente devido ao *overhead* resultante de criação de novas *threads* sempre que é efectuado um novo pedido.

Thread-pool

O modelo de concorrência *thread-pool* utiliza *threads* de um "reservatório" de modo a responder aos pedidos. Deste modo é apenas necessário criar as *threads* uma vez e podem ser reutilizadas para outros pedidos.

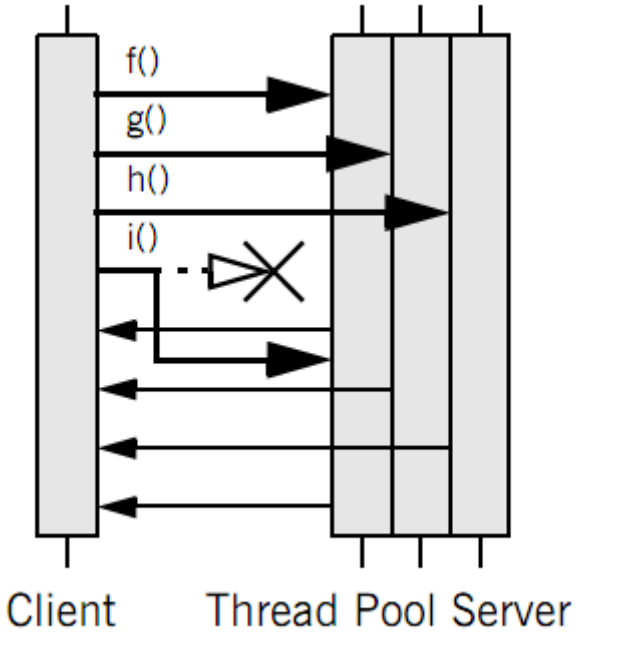

Figura 13 – Modelo de concorrência *Thread Pool*

Visto não haver tempo consumido com a criação de *threads* este modelo tem um melhor desempenho do que o modelo *thread-per-request*. O *thread-pool* é uma boa aposta se por um lado a criação e a destruição de *threads* trouxer resultados de desempenho inaceitáveis, no entanto atrasar a execução concorrente da chamada a um método também não é desejável.

3.5 Data Distribution Service

O *Data Distribution Service* (DDS) [15] é uma especificação lançada pelo OMG que oferece um *middleware* para aplicações do tipo produtor/consumidor centradas em dados.

Tal como o CORBA, o DDS permite combinar várias linguagens de programação e sistemas operativos.

O DDS define um espaço de dados global virtual que pode ser acedido pelas aplicações através de plataformas heterogéneas. Seguindo um paradigma produtor/consumidor de muitos para muitos, o produtor produz/escreve informação que pode ser consumida/lida por uma aplicação consumidor, mesmo que estejam desacoplados no tempo, espaço e sincronização. Isto significa que mesmo que uma aplicação se ligue ao sistema pode subscrever e obter os dados persistentes mesmo que tenham sido emitidos antes. O esquema de comunicação geral está descrito na Figura 14**.**

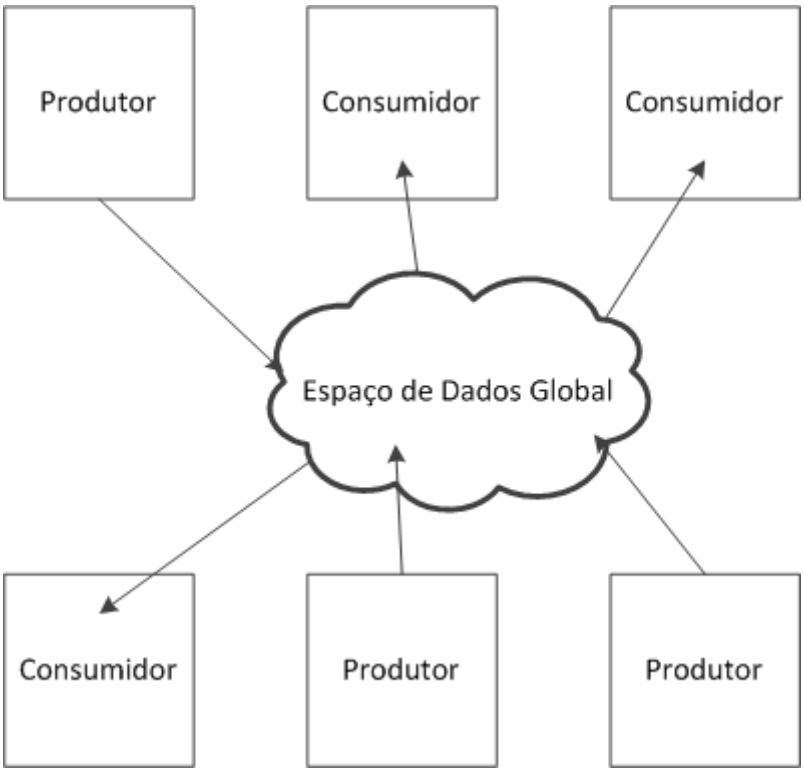

Figura 14 – Espaço Global de Dados

A unidade básica de informação de dados do DDS é o Tópico. Aplicações declaram a sua intenção de publicar Tópicos enquanto que os consumidores subscrevem tópicos num determinado domínio. Esta relação é ilustrada na Figura 15 – Domínio

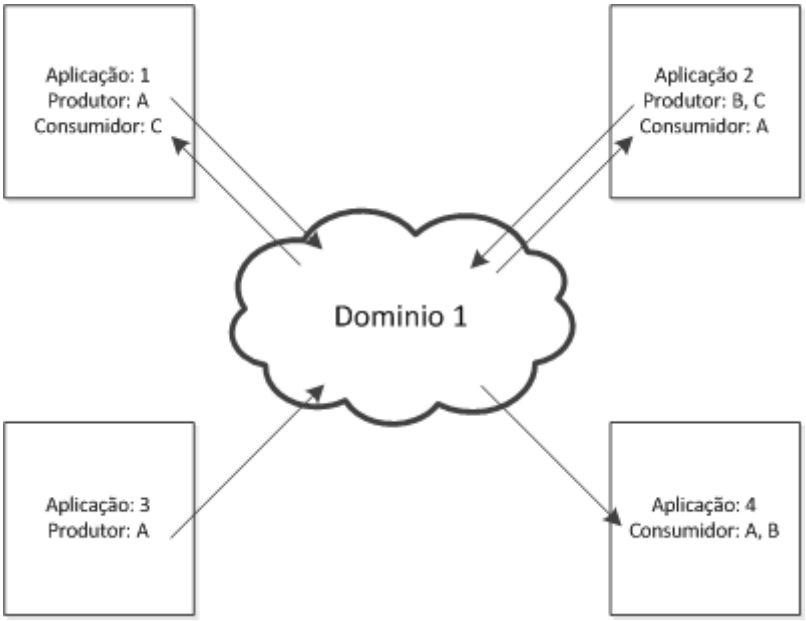

Figura 15 – Domínio

DDS define um modelo chamado *Data-Centric Publish-Subscribe* (DCPS), uma API baseada em tópicos que envolve as seguintes entidades:

- Tópico: Unidade básica de informação, que liga atomicamente o produtor e o consumidor, identificado por um nome. Permite comunicação anónima e transparente. Cada instância é um objecto definido num IDL e distinguíveis por uma chave.
- Domínio: Contexto da comunicação que fornece um ambiente virtual, isolando diferentes interesses optimizando assim a comunicação.
- Participante do Domínio: Entidade participante de um domínio.
- *Data Writer*: Entidade que pretende publicar um Tópico, fornecendo operações para escrita/envio de dados.
- *Data Reader*: Entidade que pretende consumir um Tópico, fornecendo operações para leitura/recepção de dados.
- Produtor: Entidade criada pelo Participante do Domínio para gerir um grupo de *Data Writers*.
- Consumidor: Entidade criada pelo Participante do Domínio para gerir um grupo de *Data Readers*.

A Figura 16 – Relações entre componentes exemplifica a interacção entre as entidades descritas anteriormente.

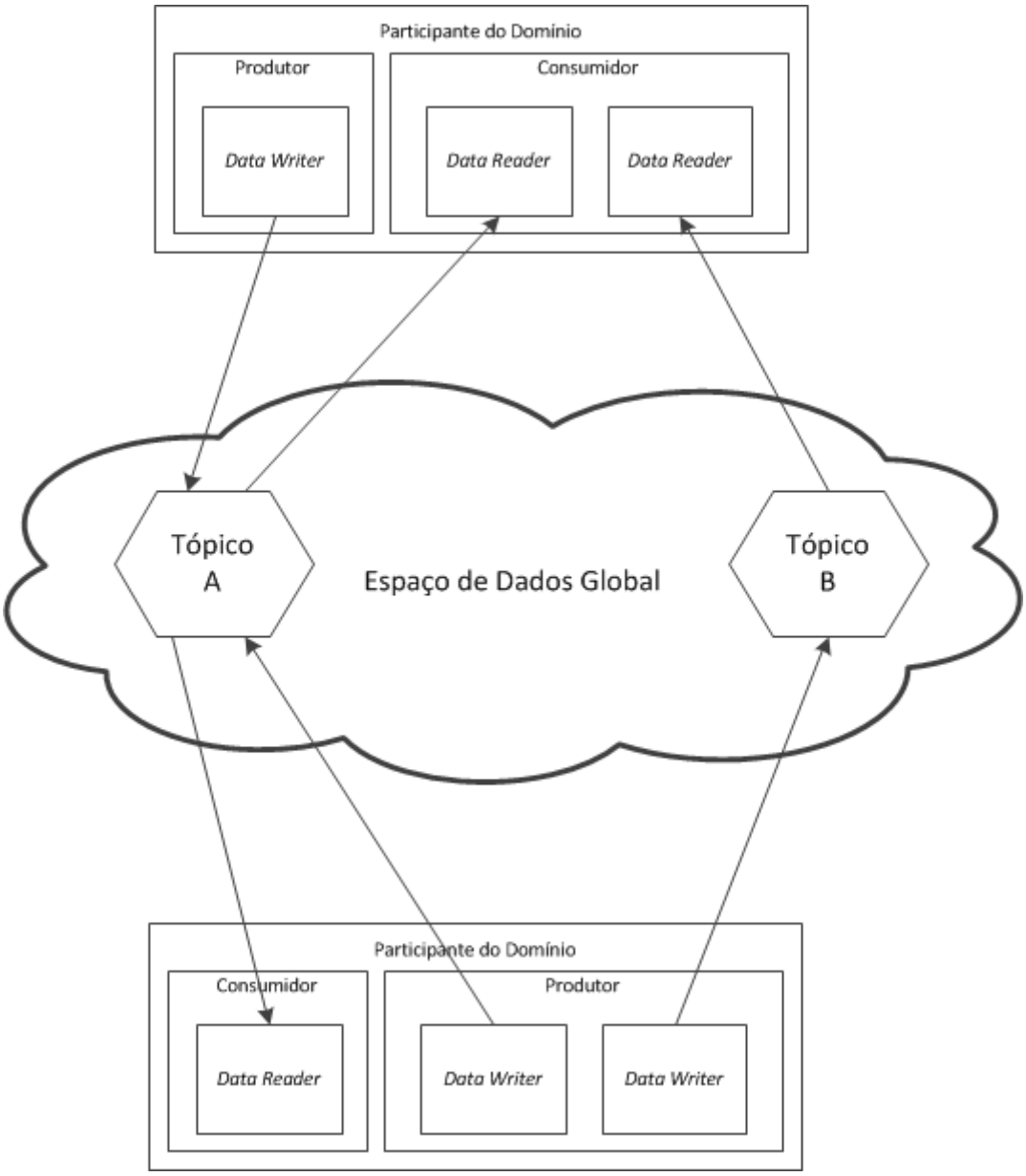

Figura 16 – Relações entre componentes

3.6 Sumário

Este capítulo abordou o trabalho existente que esteja relacionado com o projecto ou que seja opção para o mesmo. Sendo assim o capítulo começa por descrever o *middleware Basic System* e uma ferramenta utilizada para criar e gerir as bases de dados do *Basic System* ferramenta essa com o nome de *Data Store Manager*.

De seguida é apresentada a especificação CORBA assim como um caso específico de uma implementação da especificação o ORBacus. São também descritos os modelos de concorrência que a implementação ORBacus oferece.

Por ultimo é descrita a especificação *Data Distribution Service* sendo que esta especificação é uma alternativa ao canal de eventos do CORBA.

O próximo capítulo ira apresentar o trabalho realizado sendo que é visível as várias fases do modelo de desenvolvimento de software apresentado.

Capítulo 4

Trabalho Realizado

Este capítulo descreve o trabalho realizado ao longo do projecto. A apresentação será de acordo com o modelo em V, um dos modelos de desenvolvimento de software utilizados na NAV Portugal E.P.E..

4.1 Identificação de Actores

Um actor [16] em UML especifica o papel que um utilizador ou um sistema desempenha perante o sistema já existente. Neste trabalho foram identificados dois actores principais: o servidor CORBA e o cliente CORBA.

O servidor CORBA é o responsável por oferecer a interface com o DSM permitindo ao cliente CORBA utilizar as funções por ele disponíveis. A responsabilidade deste cliente é demonstrar e verificar o funcionamento desta interface criando assim a prova de conceito.

4.2 Levantamento de Requisitos

Os requisitos apresentados pelo cliente foram:

1. Utilização de CORBA para a criação da interface
2. Extensão do DSM.
3. Alterações efectuadas pelos clientes são repercutidas pela Base de Dados e pelos restantes clientes.

No entanto foi necessário fazer o levantamento tanto de requisitos funcionais como não funcionais do sistema de modo a perceber a funcionalidades que o sistema oferece bem como perceber o seu comportamento desejado.

4.2.1 Requisitos Funcionais

Foram identificados os seguintes requisitos funcionais:

REQ 1 — O servidor CORBA deverá ser integrado no *Basic System* de forma a conseguir aceder às bases de dados controladas pelo DSM.

REQ 2 — O servidor CORBA deverá ser independente do *Basic System*, sendo que uma falha no servidor CORBA não deve causar uma falha no *Basic System*.

REQ 3 — O cliente CORBA deverá conseguir efectuar as operações sobre as bases de dados controladas pelo DSM sem ter que utilizar o *Basic System*.

REQ 4 — O cliente CORBA deverá conhecer a localização da interface de modo a conseguir efectuar as operações fornecidas por este.

REQ 5 — O cliente CORBA deve ser informado sempre que houver alguma alteração sobre algum elemento da base de dados.

REQ 6 — Quando as alterações são disseminadas, todos os clientes devem receber a mesma informação, não podendo haver clientes a receber informação diferente para a mesma alteração.

REQ 7 — Após processarem a alteração dissipada todos os clientes devem ficar com a mesma vista sobre os objectos da base de dados.

REQ 8 — Cada alteração efectuada sobre um objecto da base de dados deve resultar numa dissipação de informação pelos clientes CORBA.

4.2.2 Requisitos não funcionais

Os requisitos não funcionais identificados foram os seguintes:

REQ 9 — Interoperabilidade: O *Basic System* e o CORBA devem comunicar entre eles. Os clientes devem conseguir comunicar com o *Basic System* de uma forma transparente.

REQ 10 — Capacidade: O servidor CORBA deverá permitir tantas ligações a clientes CORBA como o DSM permite.

REQ 11 — Extensibilidade: Deverá ser possível adicionar funcionalidades CORBA sem que para isso seja necessário efectuar alterações na implementação do *Basic System*.

4.3 Abordagem ao problema

De forma a conseguir satisfazer os requisitos apresentados anteriormente foi necessário escolher um ORB para a implementação, bem como os serviços CORBA a utilizar e as Politicas de Qualidade de Serviço disponíveis pelo ORB escolhido.

4.3.1 CORBA vs. DDS

Uma das opções estudadas para a abordagem ao problema para além do CORBA foi o DDS. O DDS é um *middleware* que segue o modelo produtor consumidor centrado em dados e oferece-nos propriedades desejáveis. Sendo um modelo centrado em dados conseguimos obter desacoplamento temporal e persistência de dados, que com a aproximação CORBA não é possível. Além disso com o DDS apenas é necessário definir a estrutura de cada tópico, enquanto que com o CORBA é necessário definir uma interface diferente para cada tipo de dados.

Além do DDS ser uma escolha favorável ao desenvolvimento deste projecto o objectivo era perceber o funcionamento entre o CORBA e o *Basic System*.

4.3.2 ORB escolhido

A Tabela 2 – Comparação entre ORBs (Componentes e Linguagens)

	CORBA Services			
	Event Service	Notification Service	Name Service	Transaction Service
JacORB	✗		✓	✓
ORBacus	✓	✓	✓	
ORBix E2A	✓	✓	✗	✓
OpenORB	✓	✗	✗	✓
J2SE ORB			✓	
TAO	✗	✓	✓	

Tabela 3 – Comparacao entre ORBs (Serviços) e Tabela 4 – Comparação entre ORBs (Diversos) ilustram uma comparação entre os vários ORBs existentes. Esta comparação é feita a nível de componentes, linguagens suportadas, serviços disponibilizados entre outros.

Esta informação foi retirada de um estudo [7] que pretende avaliar e comparar os vários ORBs existentes. Esta avaliação é feita tanto a nível de funcionalidades bem como a nível de desempenho e tem como objectivo ajudar o programador a escolher o ORB mais indicado a cada situação.

	Core Components					Languages	
	BOA	POA	DII/DSI	IR	ImR	Java	C++
JacORB		✔	✘	✘	✔	✔	
ORBacus		✔	✔	✘	✔	✔	✔
ORBix E2A		✔	✔	✘	✔	✔	✔
OpenORB	✔	✔	✔	✔		✔	
J2SE ORB		✔				✔	
TAO		✔	✔	✔	✔		✔

Tabela 2 – Comparação entre ORBs (Componentes e Linguagens)

	CORBA Services			
	Event Service	Notification Service	Name Service	Transaction Service
JacORB	✘		✔	✔
ORBacus	✔	✔	✔	
ORBix E2A	✔	✔	✘	✔
OpenORB	✔	✘	✘	✔
J2SE ORB			✔	
TAO	✘	✔	✔	

Tabela 3 – Comparacao entre ORBs (Serviços)

	Miscellanea			
	Stock Example	Open Source	Development	Documentation
JacORB	✔	✔	3	2
ORBacus	✔	✔	3	2
ORBix E2A	✘	✘	1	2
OpenORB	✔	✔	4	5
J2SE ORB	✔	✔	4	1
TAO	✔	✔	5	3

Tabela 4 – Comparação entre ORBs (Diversos)

Este estudo mostrou que os vários ORBs variam nos vários pontos avaliados. A lista de funcionalidades suportada por cada fornecedor não é muito significante para a escolha de um ORB pois alguns fornecedores apenas oferecem uma pequena lista de funcionalidades mas funcionam correctamente (J2SE ORB ou OpenORB) enquanto outros oferecem uma lista maior mas estes serviços não se encontram implementados correctamente (JacORB).

Outro ponto em que os ORBs diferem é a documentação e exemplos de programação. Esta característica varia entre não haver nada (J2SE ORB ou ORBix) a haver muita informação para todos os serviços (TAO e OpenORB). A avaliação mostra também que as funcionalidades mais básicas (POA, DII, IR, *Name Service*) na maioria dos ORBs funciona correctamente quando testadas independentemente. O problema começa se usarmos ORBs diferentes de modo a testar a interoperabilidade entre eles.

Outro aspecto interessante é que o ORB comercial por defeito não se mostrou melhor ou mais completo que as versões *open source*. Embora o ORBix esteja apenas disponível em versão comercial apresenta problemas com o IR e o *Name Service.* O OpenORB, ORBacus ou TAO são versões *open source* e em alguns aspectos é melhor que o ORBix.

Visto um dos requisitos iniciais ser construir provas de conceito tanto em C++ como em Java é necessário utilizar um ORB que suporte as duas linguagens devido às várias incompatibilidades, entre ORBs diferentes, reveladas por este estudo.

Como já foi referido anteriormente, as diferenças entre as versões *open source* e a versão comercial não são assim tão significativos e como o objectivo principal é cirar uma prova de conceito, e tendo em conta as razões apresentadas anteriormente a escolha recaiu sobre o ORBacus.

4.3.3 Serviços Utilizados

Seguidamente serão apresentados os serviços a utilizar para se concretizar as tarefas de disponibilização da interface remota e disseminação de dados.

Disponibilização de Interface Remota

De modo a permitir a um cliente externo ao *Basic System* aceder às bases de dados por este controladas, é necessário criar uma interface que disponibilize as funcionalidades que a API do DSM disponibiliza, sendo necessário também disponibilizar esta interface ao cliente de modo a que este a possa utilizar.

Foi utilizado o serviço *Naming Service* do CORBA para registar a interface que fornece ao cliente as funções que lhe permite manipular a Base de Dados DSM. A referência a este objecto remoto é registada no *Naming Service* pelo servidor CORBA que por outro lado é uma tarefa *Basic System* de modo a conseguir utilizar a API fornecida pelo servidor DSM. É também responsabilidade do servidor CORBA implementar a interface recorrendo a isso à API do DSM.

Em relação ao cliente este apenas necessita de conhecer a localização do *Naming Service* de modo a conseguir obter a referência a esta interface. Assim, o cliente desconhece por completo a existência do *Basic System* mas, no entanto consegue operar sobre as suas bases de dados, tornando assim transparente a existência do *Basic System*.

De seguida é apresentado os casos de uso referente a esta funcionalidade.

Disseminação de dados

De modo a replicar os dados para os restantes clientes foi utilizado o Serviço de Eventos do CORBA.

Sendo que a escolha recaiu sobre o serviço de Eventos do CORBA foi necessário de escolher qual o modelo de disseminação utilizar, as hipóteses possíveis eram o modelo *push* e o modelo *pull*. O modelo *push* é do tipo *Publisher-Subscriber* e é uma forma de comunicação de um-para-muitos, ou seja, uma das aplicações (produtor) publica uma mensagem no canal de eventos e todas as aplicações subscritoras recebem a mensagem. Este modelo é assíncrono pois quem publica os eventos não necessita de esperar pelas respostas de quem subscreve. Resumidamente, neste modelo os dados são "empurrados" do produtor para o consumidor.

No modelo *pull* os dados são "puxados" a partir dos fornecedores, não existindo a comunicação um para muitos. Neste modelo sempre que um consumidor quer consumir um evento tem de inquirir o produtor a pedir o evento, o que se torna uma desvantagem pois os consumidores podem não receber os dados no momento exacto em que eles são disseminados, podendo também haver consumidores que recebem os dados em instantes diferentes.

O modelo escolhido foi o modelo *push*, neste modelo os dados são "empurrados" do produtor para o consumidor, sendo neste caso o servidor CORBA o produtor e os clientes os consumidores, ou seja, sempre que um cliente deseja efectuar uma alteração à base de dados é enviado um pedido ao Servidor através da invocação do método do objecto remoto, sendo que este objecto é obtido através do *Naming Service*. Quando a alteração é efectuada então o servidor actualiza todos os clientes "empurrando" o evento para o canal de eventos. Os clientes por sua vez consomem o evento de modo a actualizar os seus dados localmente, incluindo o próprio cliente que efectuou a alteração. Sendo assim, se os clientes tiverem todos o mesmo estado, ao receberem o

evento todos efectuam a mesma operação mantendo assim a coerência entre clientes e servidor.

A escolha recaiu neste modelo pois o objectivo é que seja o servidor a informar os clientes quando as operações são realizadas, caso fosse escolhido o modelo *pull* tinha de ser os clientes a pedir a informação ao servidor, podendo haver alturas em que os dados visíveis nos clientes não estavam coerentes com os dados do servidor. Também haveria uma sobrecarga na rede, devido aos pedidos *pull* efectuados pelos clientes.

4.3.4 Modelo de Concorrência

O DSM está construído de forma a atender individualmente a cada pedido, isto significa à medida que os pedidos vão chegando vai havendo uma espera para que o pedido mais recente seja atendido. Deste modo o DSM tem um comportamento bloqueante cada vez que executa um pedido que estivesse em espera.

Sendo que um dos objectivos do projecto é integrar o CORBA com o *Basic System*, com o mínimo de quebra de desempenho do BS, a opção escolhida para modelo de concorrência a utilizar no servidor CORBA foi a de *Thread-per-client*.

Com este modelo e tendo em conta o que é explicado na secção 3.4.1 deste documento, o servidor CORBA, ao contrário do DSM, não terá um comportamento bloqueante, o que permitirá com que o único tempo de espera que os pedidos se encontram sujeitos continua a ser o que já existia. Sendo assim o modelo de concorrência da integração do CORBA com o BS será o seguinte.

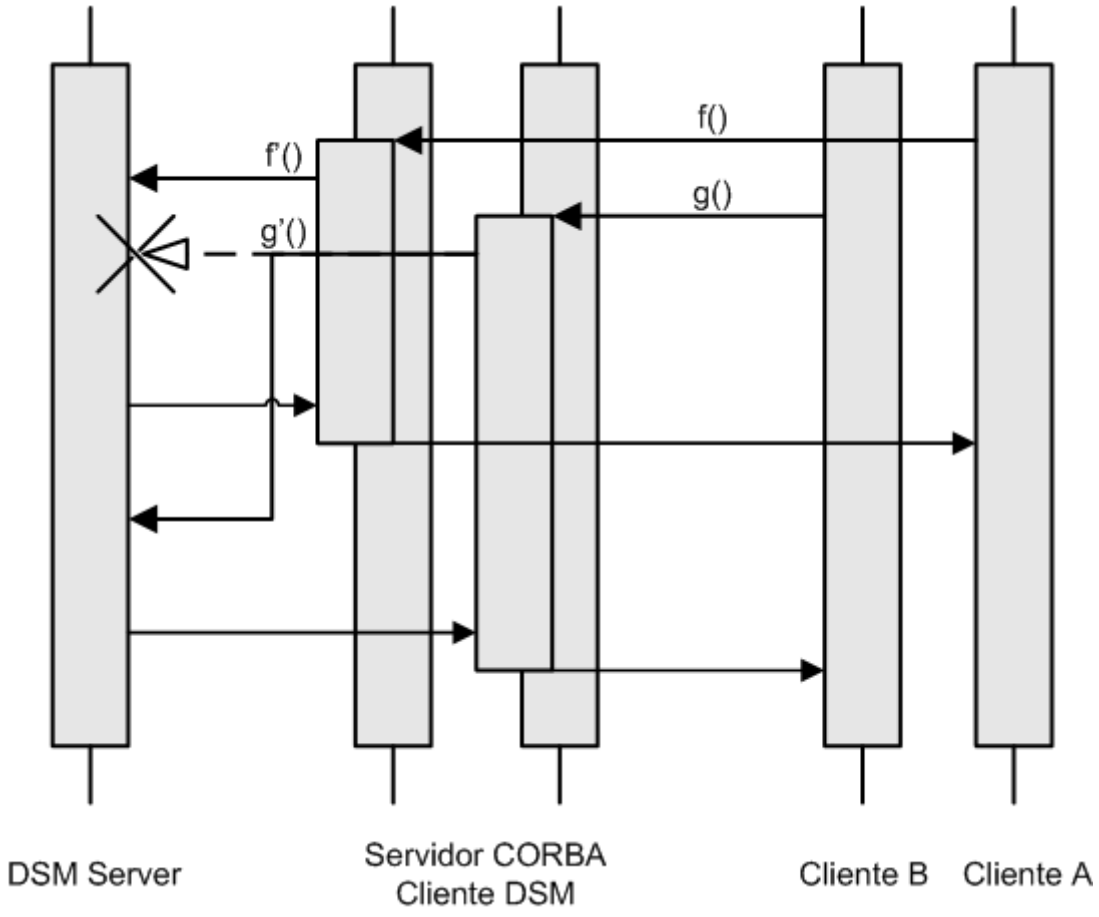

Figura 17 – Modelo de Concorrência

Como é visível na Figura 17 a única espera que acontece é no servidor de DSM, e esta espera já acontecia tal como foi referido, deste modo o atraso causado pela adição da camada de CORBA é mínimo, sendo esta a razão para a escolha efectuada.

4.4 Casos de Uso

Após a identificação dos actores do sistema, o levantamento de requisitos funcionais e não funcionais e da análise anterior foi construído um diagrama de casos de uso e feita a descrição de cada um.

4.4.1 Diagrama Casos de Uso

Foram identificados os seguintes casos de uso:

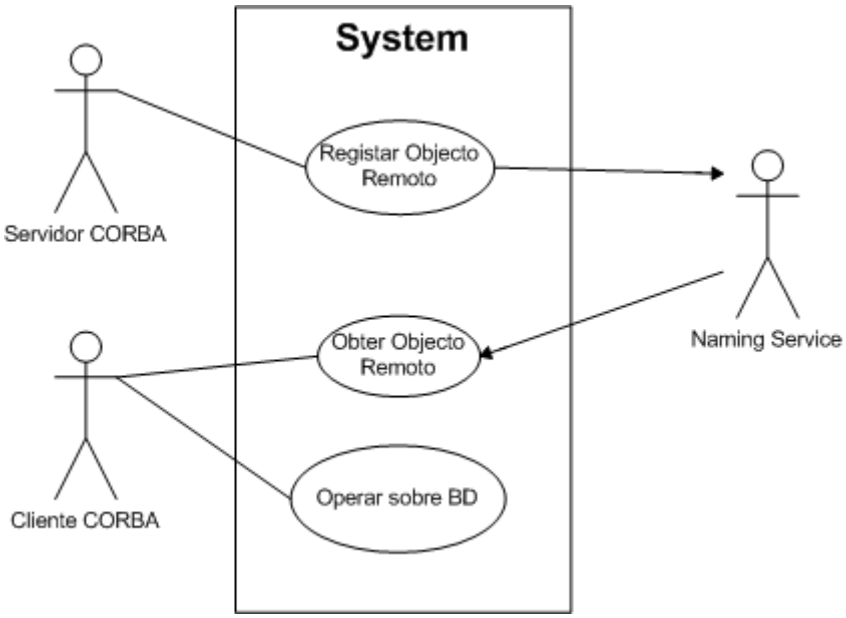

Figura 18 – Diagrama de Casos de Uso

4.4.2 Descrição de Casos de Uso

Tendo em conta o diagrama da Figura 18 – Diagrama de Casos de Uso são seguidamente apresentadas as descrições de dois casos de uso. São eles o "Registar Objecto Remoto" e "Obter Objecto Remoto". O caso de uso com o nome "Operar sobre BD" não é descrito pois não havia nenhuma mais valia em apresentar uma descrição.

Caso de Uso: Registar Objecto Remoto

Actor Principal: Servidor CORBA

Pré-Condições:

- O Servidor CORBA necessita de conhecer a localização do *Naming Service.*
- O *Naming Service* necessita de estar a executar.

Pós-Condições:

- A referência do objecto remoto é registada com sucesso no *Naming Service*.

Cenário Principal de Sucesso:

1. O Servidor CORBA obtém a referência do *Naming Service*
2. O Servidor CORBA cria um nome.
3. O Servidor CORBA cria uma instância do objecto remoto.

4. O Servidor CORBA associa o nome criado em 2 e associa-o ao objecto criado em 3.

Caso de Uso: Obter Objecto Remoto

Actor Principal: Cliente CORBA

Pré-Condições:

- O Cliente CORBA necessita de conhecer a localização do *Naming Service.*
- O *Naming Service* necessita de estar a executar.
- A referência ao objecto remoto tem de ser registada com sucesso do *Naming Service.*

Pós-Condições:

- O cliente obtém a referência do objecto remoto.

Cenário Principal de Sucesso:

1. O Cliente CORBA obtém a referência para o *Naming Service.*
2. O Cliente CORBA cria um nome.
3. O Cliente CORBA obtém a referência ao objecto remoto através do nome criado em 2.
4. O Cliente CORBA efectua um *downcast* de modo a pode invocar os métodos disponibilizados pelo objecto remoto.

4.5 Desenho

Como foi dito anteriormente a comunicação entre cliente e servidor DSM é feita com o auxílio de tarefas *Basic System*. É necessário estender esta comunicação com o apoio do CORBA de modo a permitir fornecer a clientes externos a possibilidade de operar sobre as bases de dados controladas para o DSM sem que estes terem de recorrer ao *Basic System*.

Para isso é necessário criar uma interface que mapeie todas as funções do DSM, sendo que esta interface deve estar disponível ao cliente para que este tenha conhecimento das funções a utilizar. De forma a disponibilizar a interface ao cliente é utilizado o *Naming Service*, disponibilizado pelo CORBA. É recorrendo a este serviço que é registada a localização da interface de modo a que o cliente consiga obter uma referência à mesma. Com a utilização deste serviço disponibilizado pelo CORBA o

cliente apenas necessita de conhecer a localização do *Naming Servic*e e o nome da interface.

Na Figura 19 – é apresentado um diagrama de fluxo de dados entra as componentes funcionais que fazem parte da integração do CORBA com o *middleware* existente. Nesta figura distingue-se a barreira entre *Basic System* e o exterior, sendo o exterior a camada CORBA criada.

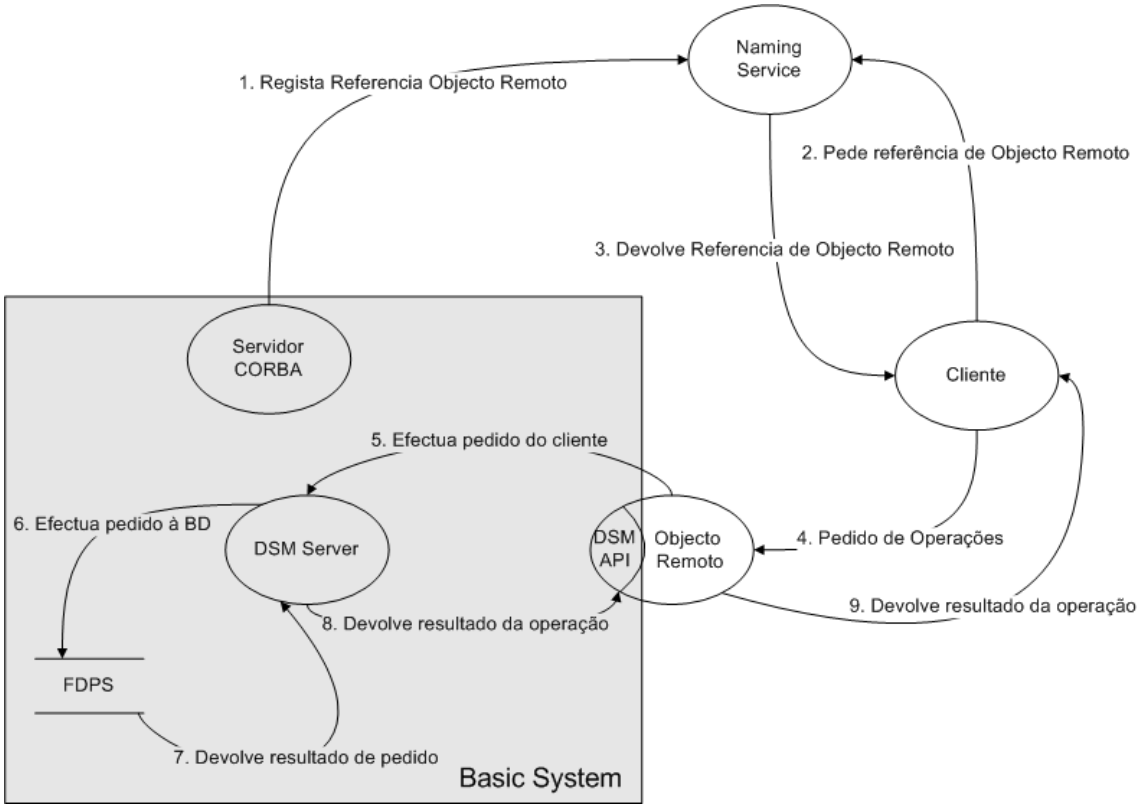

Figura 19 – Diagrama de Fluxo de Dados CORBA – Basic System

A Figura 19 – Diagrama de Fluxo de Dados CORBA – Basic System tem identificado o fluxo de dados existente entre as componentes CORBA e as componentes *Basic System*, sendo que a parte sombreada corresponde aos componentes assentes em *Basic System*. A única componente existente é o Servidor DSM e a API fornecida por este. É necessário fazer de raiz tanto o Servidor e Cliente CORBA bem como a implementação do objecto remoto a disponibilizar. O Naming Service é disponibilizado pelo ORB escolhido.

Este diagrama tem identificado cada fluxo de dados existente com um número, esta identificação corresponde à ordem como é feita uma interacção com o *middleware* utilizando o CORBA.

Seguidamente é apresentado um diagrama de sequência que mostra de uma forma mais clara como é feita essa interacção com o *Basic System*.

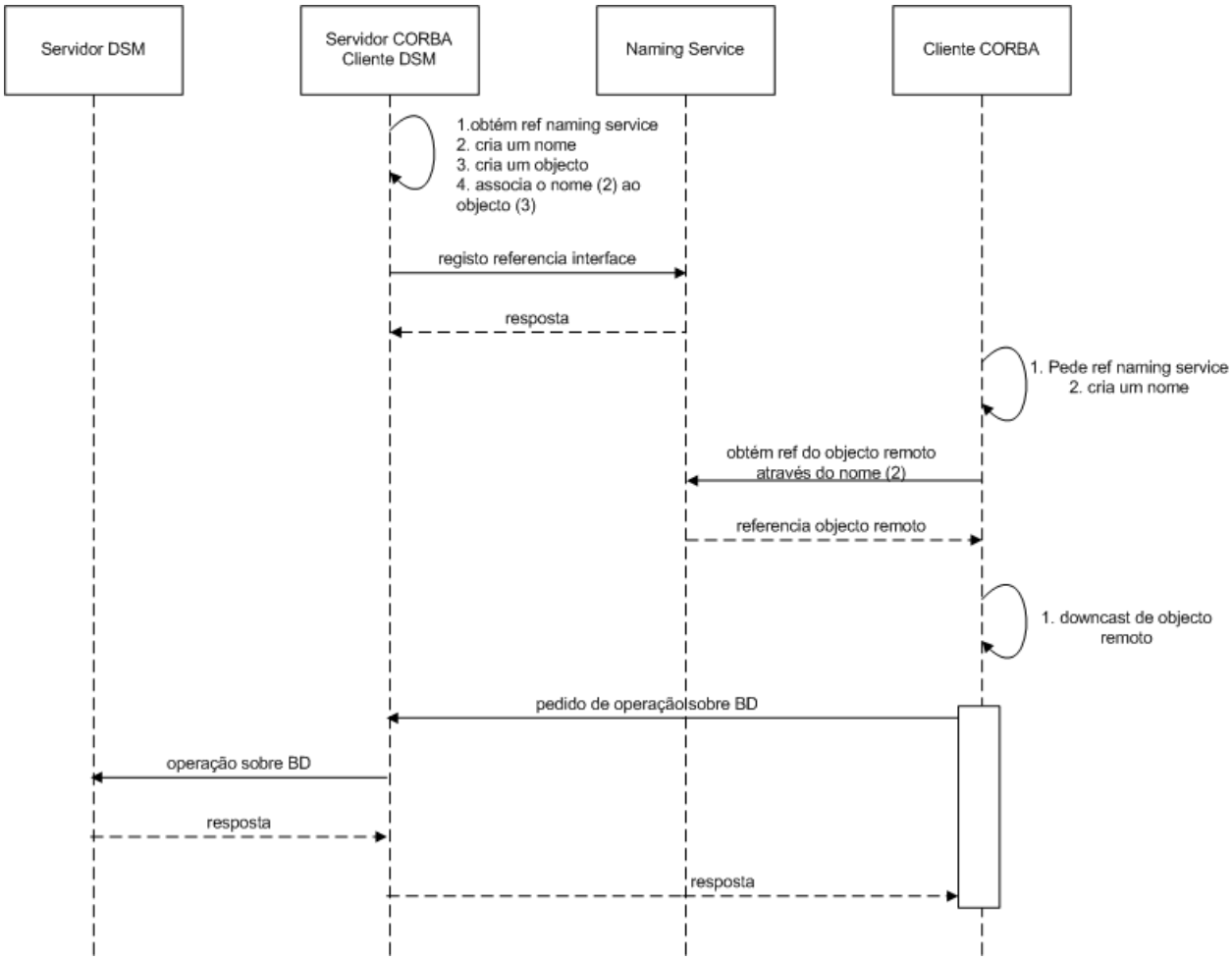

Figura 20 – Diagrama de sequência invocação remota

Este diagrama pretende ilustrar de uma forma mais clara para o leitor todos os passos que são necessários para que um cliente que não tenha conhecimento da existência do *Basic System* consiga efectuar uma operação no mesmo.

Identificado como requisito foi também a disseminação de alterações por todos os clientes. A opção para concretizar este requisito recaiu sobre a utilização do *Event Service* do CORBA, tal como é explicado na secção 4.3.3

Seguidamente é apresentado um diagrama que mostra a forma como, ao ser efectuada uma alteração na base de dados por um cliente, essa alteração é disseminada para todos os clientes.

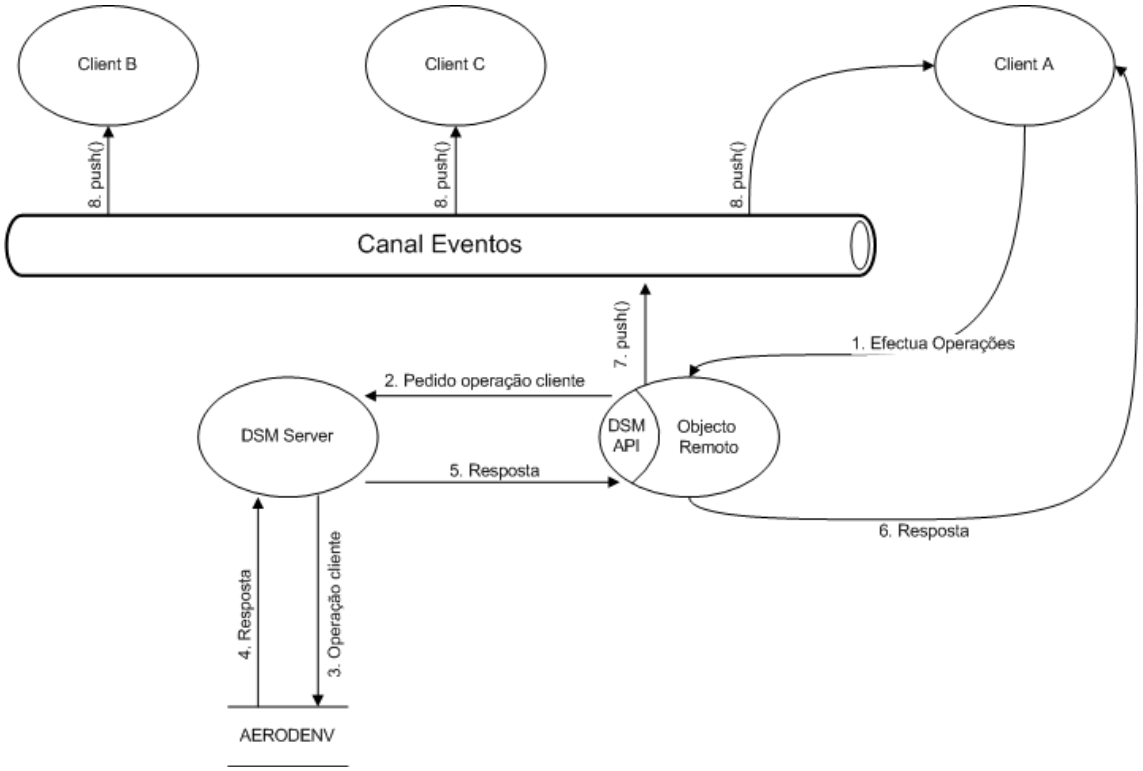

Figura 21 – Diagrama de fluxo de dados – Canal de Eventos

Com esta aproximação todos os clientes se encontram ligados a um canal de eventos, canal esse que serve para disseminar as alterações efectuadas sobre a base de dados.

É visível que o Cliente A efectua uma operação remota sobre a base de dados, esta operação é feita recorrendo ao *Naming Service*, tal como já foi explicado anteriormente. Após a realização desta operação o objecto remoto é responsável por disseminar um evento para o canal de eventos, sendo este objecto remoto o produtor. Os clientes são todos consumidores de eventos e encontram-se à espera de eventos que entrem no canal.

Sendo este o desenho proposto para a concretização de uma solução do problema, a próxima secção apresenta a forma como este desenho foi implementado.

4.6 Implementação

Esta secção tem como objectivo abordar o trabalho realizado no âmbito da implementação. Sendo assim será referido todo o trabalho que tenha envolvido produção ou alteração de software. Toda a implementação foi feita sobre 120012-14 SunOS 5.10 e foi utilizado o SunStudio [18] para compilar o código desenvolvido em C++.

4.6.1 Data Store Manager – Data Description Compiler

Como foi dito anteriormente – secção 3.2.1 – o DSM contém uma ferramenta que permite compilar os ficheiros que contêm a descrição das bases de dados de modo a gerar os ficheiros necessários para a interacção com esta. Um dos ficheiros gerados é um ficheiro IDL mas que apenas contem o mapeamento das estruturas dos objectos da base de dados.

Foi adicionado a este compilador a funcionalidade de gerar o IDL com as funções necessárias para interagir com a Base de Dados. Desta forma é possível utilizar este ficheiro gerado e compilar com um compilador IDL de modo a gerar os *stubs* necessários para implementar a interface CORBA.

Optou-se por manter a assinatura dos métodos tal qual a API do DSM, ou seja o nome das funções e os atributos recebidos por cada uma delas são iguais tanto no IDL como no API do DSM, de modo a diminuir a entropia.

Visto o ambiente onde se enquadra o projecto ser bastante complexo, e as estruturas das bases de dados conterem bastantes valores, a evolução do compilador existente permitir gerar o ficheiro IDL compatível com CORBA e contendo todas as funções e estruturas necessárias para interagir com as bases de dados é um passo bastante importante pois torna a tarefa de criação do ficheiro IDL bastante mais simples e optimizada.

4.6.2 *Naming Service*

Visto tratar-se de uma prova de conceito foi criada uma base de dados que foi povoada com dados fictícios de modo a simplificar a implementação, e deste modo poder dedicar mais atenção à integração do CORBA com o *Basic System* e não à criação/manutenção dos dados de forma coerente, visto essas tarefas serem bastante complexas pois o conteúdo dos dados em questão é bastante extenso e complexo.

Este ficheiro foi compilado com o DDC evoluído durante a realização do projecto, gerando assim o IDL reconhecido pelo CORBA e pronto a ser compilado.

Ao ser gerado o IDL pela ferramenta DDC (secção 4.6.1 é necessário recorrer a uma ferramenta fornecida pela distribuição CORBA utilizada, o ORBacus, tendo como objectivo compilar o IDL de forma a gerar os adaptadores tanto para o lado do servidor como para o lado do cliente. Estes adaptadores são os responsáveis por efectuar a interoperabilidade entre clientes e servidor, permitindo a comunicação entre ambos. Os

adaptadores são gerados na linguagem C++ e Java, sendo que os adaptadores na linguagem Java apenas servem para criar clientes Java.

Associada ao IDL surge também uma interface contendo as funções definidas no mesmo. Esta interface é implementada do lado do servidor na linguagem C++, pois o objectivo da interface é utilizar a biblioteca DSM que também se encontra implementada na mesma linguagem. O servidor também é implementado em C++ pois será executado como uma tarefa do *Basic System*.

As responsabilidades do servidor CORBA são as de obter a referência para o *Naming Service*, criar uma instância do objecto que implementa a interface obtida da compilação do IDL e por último efectuar o registo deste objecto no *Naming Service* de modo a que esse objecto possa ser acedido pelos clientes que o desejarem.

O servidor precisa de efectuar os seguintes passos:

1. Inicializar o ORB e o *Root* POA – É necessário obter a referência para estes dois objectos para se poder, inicializar, activar e registar o objecto remoto.

   É na fase de iniciação o ORB que é configurado o modelo de concorrência a utilizar. Essa configuração é feita através da configuração da propriedade de modelo de concorrência do ORB.

```
props -> setProperty("ooc.orb.oa.conc_model", "thread_per_client");
ORB_var orb = OBCORBA::ORB_init(argc, argv, props);
Object_var poaObj = orb->resolve_initial_references("RootPOA");
POA_var rootPOA = POA::_narrow(poaObj.in());
```

2. Criar e activar o objecto remoto – O objecto remoto necessita ser activado antes de algum cliente o poder utilizar. A activação do objecto remoto associa o objecto ao POA e a um identificador único. Este identificador é retornado pela função *CORBA::POA::activate_object()* e é automaticamente gerado pelo *Root* POA.

```
Dsm_impl* impl = new Dsm_impl(argv[1]);
ObjectId_var myObjID = rootPOA -> activate_object(impl);
Object_var o = rootPOA->servant_to_reference(impl);
```

3. Obter referência para o *Naming Service* – A referência inicial é obtida invocando *CORBA::ORB::resolve_initial_references("NameService")*

```
Object_var obj = orb -> resolve_initial_references("NameService");
NamingContext_var nc = CosNaming::NamingContext::_narrow(obj.in());
```

4. Criar um Nome – Este nome é criado de modo a ser associado à referência do objecto remoto, para de seguida ser registado no *Naming Service*.

```
CosNaming::Name implName;
implName.length(1);
implName[0].id = CORBA::string_dup("dsmImpl");
implName[0].kind = CORBA::string_dup("");
```

5. Registar o objecto remoto no *Naming Service* – A referência ao objecto é publicada no *Naming Service*, sendo criada uma ligação que associa o nome do objecto à sua referência. Para isso é utilizada a função *CosNaming::NamingContext::rebind()*, esta função pode criar uma nova ligação com o Nome n, ou se uma ligação com este Nome já existir rescreve a mesma. Torna-se conveniente se houver necessidade de actualizar as ligações aos objectos sempre que o servidor é iniciado.

```
nc -> rebind(implName, o);
```

6. Activar o POA *Manager* – Esta activação permite invocar pedidos aos objectos POA.

```
manager -> activate();
```

7. Lançar a execução do ORB – Neste ponto a inicialização do servidor encontra-se completa. Ao lançar a execução do ORB passa-se o controlo da aplicação para o mesmo, pois a função responsável por isto é bloqueante. De modo a conseguir manter a interacção com a consola do *Basic System* foi necessário criar uma *thread* responsável pela execução

dos passos anteriores. Deste modo é possível manter o controlo já existente nesta tarefa *Basic System*.

```
orb -> run();
```

A implementação da interface, tal como já foi dito anteriormente, utiliza a biblioteca DSM, sendo assim a interface tem o comportamento de um cliente DSM criando a ponte entre o mundo *Basic System* e o exterior. Esta interface inicialmente apenas oferece ao cliente CORBA três funções do DSM, *open*, *first* e *read*, sendo que o cliente pode abrir uma base de dados, obter o primeiro elemento da mesma e ler o conteúdo desse elemento.

De forma a criar a prova de conceito foi necessário implementar um cliente CORBA que obtém a referência para o *Naming Service* e de seguida obtém a referência para o objecto remoto registado pelo servidor CORBA.

Os passos que o cliente necessita executar são os seguintes:

1. Inicializar o ORB – A inicialização do ORB é feita da mesma maneira que no servidor.

```
ORB_var orb = ORB_init(argc, argv);
```

2. Obter a referência para o *Naming Service* – Esta referência é obtida da mesma maneira que no servidor.

```
Object_var obj = orb -> resolve_initial_references("NameService");
NamingContext_var nc = CosNaming::NamingContext::_narrow(obj.in());
```

3. Obter a referência para o objecto remoto – A referência para o objecto remoto é obtida do *Naming Service* resolvendo o Nome previamente criado e registado pelo servidor. É necessário fazer *downcast* do objecto obtido de modo a conseguir invocar remotamente as funções fornecidas.

```
Name aName;
aName.length(1);
aName[0].id = CORBA::string_dup("dsmImpl");
aName[0].kind = CORBA::string_dup("");
Object_var aObj = nc -> resolve(aName);

Dsm_var dsm = fpl::functions::Dsm::_narrow(aObj.in());
```

4. Invocar funções remotas – Utilizar a referência do objecto remoto de modo a conseguir invocar as funções desejadas.

```
dsm -> open("fpl");
```

Este cliente foi desenvolvido em Java e C++ sendo que cada uma das implementações apenas conhece a interface proveniente da compilação do IDL.

Ao obter a referência do objecto remoto com sucesso o cliente efectua a chamada das funções fornecidas pela interface e ao mesmo tempo essa chamada é feita no *Basic System* sendo que a utilização do *middleware Basic System* fica transparente para o cliente. Com a implementação do cliente nas duas linguagens é possível verificar interoperabilidade entre linguagens que o CORBA oferece.

4.6.3 Canal de Eventos

Esta secção pretende apresentar os detalhes de implementação do canal de eventos. A implementação que vai ser referida de seguida não ficou completamente concretizada devido ao tempo para realização do estágio ter terminado. Mesmo não estando totalmente concretizada são apresentados os detalhes mais importantes de forma a se perceber o que tem de ser feito.

O canal de eventos no contexto deste projecto tem o objectivo de disseminar por todos os clientes que se encontrem ligados ao servidor DSM as alterações feitas nas bases de dados.

Inicialmente é preciso definir qual o tipo de canal pretendido, sendo que o canal pode ser *typed* (é conhecido em tempo de compilação o tipo dos objectos que são inseridos no canal) ou *untyped* (o tipo de objectos inseridos no canal não é conhecido). No ambiente em que o projecto se integra o tipo de canal escolhido é *typed* pois assim consegue-se ter melhor controlo sobre os dados que são disseminados.

Como é referido na secção 4.3.3 vai ser usado um modelo *push*. Ao utilizar este modelo é necessário implementar o consumidor, sendo que o produtor apenas necessita estar conectado ao canal e inserir os dados no canal.

Sendo que quem opera directamente nas bases de dados controladas pelo servidor DSM é o objecto remoto, é este o responsável por ser o produtor *push*. Visto isto quando este objecto é criado é necessário obter a referência para o canal de eventos, e

sempre que é efectuada alguma operação que altere o estado de algum objecto na base de dados é invocado o método *push,* que insere dados no canal.

Como o objectivo é que todos os clientes conectados consumam os eventos e se mantenham consistentes com o servidor, é necessário que todos estendam a classe abstracta do consumidor *push* e implementem a função *push*. Visto isto sempre que o produtor insere dados no canal invoca a função *push*. O consumidor tem o comportamento definido na implementação da função com o mesmo nome.

De modo a que os clientes consigam continuar a operar quando recebem informação disseminada pelo canal de eventos, é necessário criar uma nova *thread* responsável por tratar esta informação.

4.7 Testes

Esta secção pretende apresentar os testes realizados no âmbito do projecto. Foram realizados testes de verificação de requisitos e testes de desempenho. Os testes de desempenho mostraram-se bastante importantes pois são estes testes que mostram o desempenho da camada CORBA integrado com o *Basic System*.

4.7.1 Testes de Verificação de Requisitos

Os testes de verificação de requisitos foram realizados localmente, na máquina onde o projecto foi desenvolvido.

A Tabela 5 – Verificação de requisitos apresenta os resultados obtidos para os requisitos funcionais identificados na secção 4.2.1 .

ID	Descrição do Requisito	Caso de sucesso	Resultado
REQ 1	Integrar Servidor CORBA no *Basic System* para aceder à Base de Dados	O Servidor CORBA é executado como sendo uma tarefa *Basic System*	OK
REQ 2	Criar independência entre o Servidor CORBA e o *Basic System* de modo a que uma falha no Servidor CORBA	O Servidor CORBA crasha mas o *Basic System* continua a funcionar	OK
REQ 3	Permitir ao cliente interagir com a Base de Dados controlada pelo	O Cliente CORBA consegue obter um objecto da Base de	OK

	DSM sem utilizar o *Basic System*	Dados	
REQ 4	Permitir ao cliente conhecer a localização do objecto remoto	O Cliente CORBA consegue obter com sucesso a referencia para o Objecto Remoto	OK
REQ 5	Informar o cliente CORBA sempre que houver alguma alteração sobre algum elemento da base de dados	O cliente CORBA A faz uma alteração sobre um objecto na Base de Dados e o cliente CORBA B recebe a alteração efectuada.	Não Verificado
REQ 6	Todos os clientes devem receber a mesma informação aquando as alterações forem disseminadas	Os dados que chegam a todos os clientes são os mesmos.	Não Verificado
REQ 7	Todos os clientes CORBA devem ter a mesma vista depois de efectuarem as alterações dissipadas	A vista de todos os clientes é igual após efectuarem as alterações recebidas	Não Verificado
REQ 8	Cada alteração efectuada sobre um objecto da base de dados deve resultar numa dissipação de informação pelos clientes CORBA.	O cliente CORBA A faz uma alteração e o servidor tem de efectuar um *push*.	Não Verificado

Tabela 5 – Verificação de requisitos

A tabela contém requisitos que não foram validados pois tal como é referido na secção 4.6.3 a implementação que permitiria verificar os requisitos em causa não foi efectuada.

4.7.2 Testes de Desempenho

Visto a prova de conceito ter sido um sucesso, e ambos os clientes terem conseguido operar sobre a base de dados foi necessário estudar a sobrecarga que esta nova camada trazia. Para isso foram realizados testes de desempenho. Estes testes consistem na medição de tempo das operações tanto do lado da interface, que se comporta como um cliente DSM, como do lado do cliente CORBA. Assim é possível

fazer uma comparação dos resultados e concluir se o acréscimo de tempo é realmente significativo e se poderá prejudicar ou não o correcto funcionamento da aplicação.

De modo a poder efectuar estes testes de desempenho foram seleccionadas um conjunto de operações consideradas as principais quando falamos de manipulação de dados numa base de dados. As operações escolhidas foram:

- *open*
- *close*
- *create*
- *findFirst*
- *read*
- *update*

Foram criados um conjunto de testes automatizados com recurso à linguagem C++. O cliente CORBA é o responsável por executar iterativamente estes testes.

Para conseguir obter os dados pretendidos é necessário capturar o tempo no momento em que a função é invocada e no momento em que o valor de retorno é obtido. É necessário fazer esta captura tanto do lado cliente CORBA como do lado do interface para conseguir obter o tempo de execução de cada função.

Visto o Servidor CORBA/ Cliente DSM efectuar também capturas de tempo e o Cliente CORBA depender deste para efectuar também as suas medidas, o seu desempenho irá ser sempre afectado. No entanto essa dependência não se mostrou ser considerável.

A Figura 22 – Captura de tempos ilustra o momento em que são feitas estas medições.

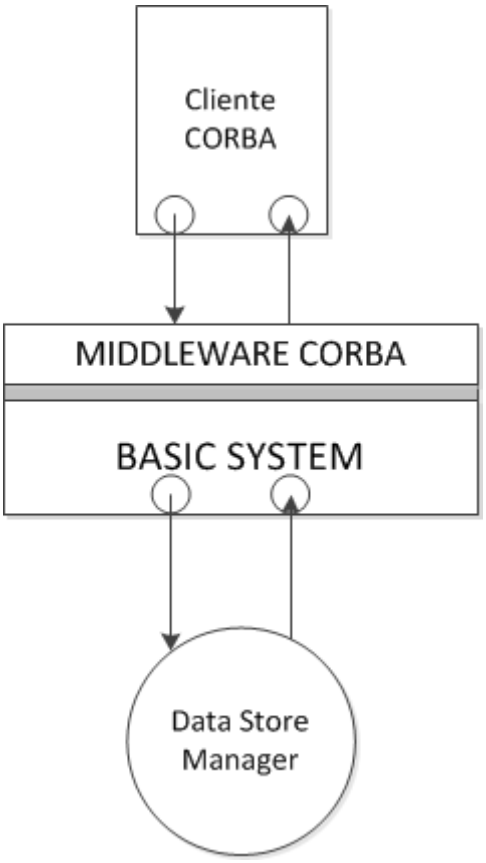

Figura 22 – Captura de tempos

Os testes executados foram os seguintes:

- Teste 1 – Abrir e fechar a Base de Dados 1000 vezes (*open* e *close*)
- Teste 2 – Ler todos os objectos da Base de Dados (*read*)
- Teste 3 – Encontrar 1000 vezes um determinado objecto na Base de Dados (*findFirst*)
- Teste 4 – Fazer update a todos os objectos da Base de Dados (*update*)
- Teste 5 – Criar 1000 objectos novos na Base de Dados (*create*)

Para testar o comportamento do sistema foi utilizada a base de dados de planos de voo FPLi. Esta base de dados contém 869 planos de voo correspondentes a uma semana de voos no período de inverno. Visto se tratar de uma base de dados diferente da utilizada foi necessário gerar um novo IDL e implementar uma nova interface bem como um novo cliente. Os testes não são mais do que a captura do momento exacto em que cada função é invocada e que uma resposta chega. Este tempo é posteriormente escrito para um ficheiro. O processo é exactamente o mesmo nas duas camadas.

É impossível tornar os testes da camada do *middleware* CORBA independentes dos testes realizados na camada *Basic System*, visto a primeira camada depender da segunda. Ou seja, os testes da camada do middleware CORBA contêm a captura de tempo e a escrita para o ficheiro dos testes da camada do *Basic System*. No entanto, de modo a determinar, se o comportamento do sistema operativo em relação à escrita para o ficheiro estava a comprometer os testes efectuados foi seguida uma segunda abordagem. Em vez de os dados serem escritos para o ficheiro sempre que eram calculados, estes foram inicialmente todos escritos para a memória e apenas no fim da execução da aplicação, foram escritos para o ficheiro. No entanto verificou-se que nas duas abordagens tomados, os dados obtidos não foram diferentes.

Com os dados obtidos foi calculado a média e o desvio padrão para cada operação. No geral o sistema comporta-se de forma regular mas em média 6.5% dos resultados foram anormais, tendo estes sido desprezados de modo a poder efectuar o estudo de performance.

Os resultados são apresentados na Tabela 6 – Média *Regular priority class* e Tabela 7 – Desvio padrão *Regular priority class* sendo que a Figura 23 – Média Cliente Regular Time apresenta o gráfico de modo a tornar a leitura mais intuitiva ao leitor.

Como se pode verificar, no geral o tempo de comunicação da camada CORBA é superior ao tempo de comunicação do *Basic System*, exceptuando na função de *create*. No entanto este resultado é o esperado pois existe processamento adicional ao criar um objecto novo na base de dados. No entanto esta nova camada, no geral, quase duplica o tempo de execução das operações.

Média (ms)						
	Close	**Create**	**Find First**	**Open**	**Read**	**Update**
DSM Client	0,154	0,371	0,125	0,138	0,122	0,237
CORBA+DSM Client	0,332	0,617	0,364	0,306	0,419	0,545
CORBA Client	0,178	0,246	0,239	0,169	0,297	0,308

Tabela 6 – Média *Regular priority class*

Desvio Padrão (ms)						
	Close	**Create**	**Find First**	**Open**	**Read**	**Update**
DSM Client	0,306	0,038	0,260	0,103	0,054	0,032
CORBA+DSM Client	0,448	0,099	0,434	0,173	0,276	0,169
CORBA Client	0,306	0,080	0,339	0,116	0,229	0,161

Tabela 7 – Desvio padrão *Regular priority class*

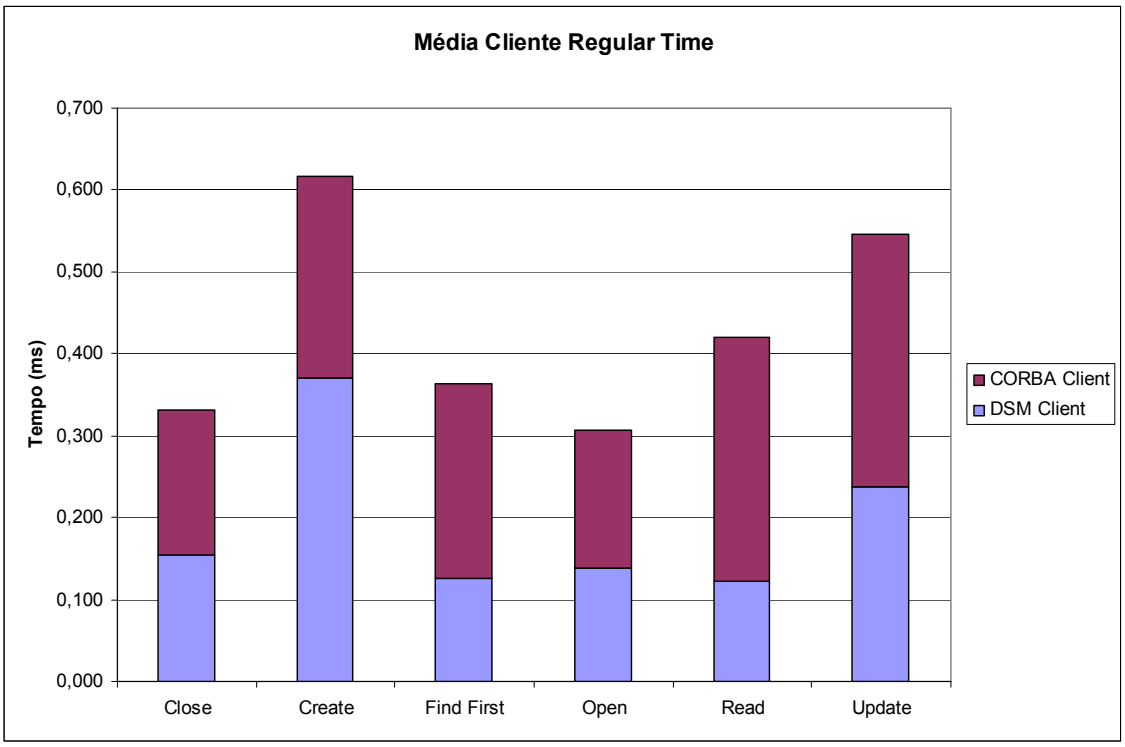

Figura 23 – Média Cliente Regular Time

No entanto, o Solaris tem classes principais de prioridade: *Real-Time*, *Regular* e *Idle*. Normalmente as aplicações são executadas em *regular priority class*, o que acontece com o cliente CORBA. O *Basic System* por sua vez é executado em *Real-Time priority class*. De modo a executar as duas componentes na mesma classe de prioridade foi utilizada uma tarefa *Basic System* que dá pelo nome de *Launcher*. Esta tarefa permite executar aplicações externas como se fossem uma tarefa *Basic System*. Assim é possível executar o cliente CORBA em *Real-Time priority class*.

Após executar todo o sistema em nível máximo de prioridade verificou-se que foi possível melhorar consideravelmente a desempenho do Cliente CORBA. No entanto os resultados são os mesmos. Na Tabela 8 – Média *Real-Time priority class* e Tabela 9 – Desvio padrão *Real-Time priority class* encontram-se os resultados calculados.

Média (ms)						
	Close	**Create**	**Find First**	**Open**	**Read**	**Update**
DSM Client	0,122	0,384	0,127	0,115	0,131	0,249
CORBA+DSM Client	0,249	0,595	0,338	0,240	0,131	0,524
CORBA Client	0,127	0,212	0,211	0,124	0,256	0,275

Tabela 8 – Média *Real-Time priority class*

Desvio Padrão (ms)						
	Close	Create	Find First	Open	Read	Update
DSM Client	0,042	0,041	0,117	0,036	0,104	0,064
CORBA+DSM Client	0,181	0,061	0,207	0,064	0,346	0,081
CORBA Client	0,173	0,028	0,169	0,043	0,326	0,034

Tabela 9 – Desvio padrão *Real-Time priority class*

A Figura 24 – Média Cliente CORBA Real Time priority class apresenta os dados contidos nas duas últimas tabelas de forma mais perceptível sendo que se pode comprovar a melhoria dos resultados do cliente CORBA.

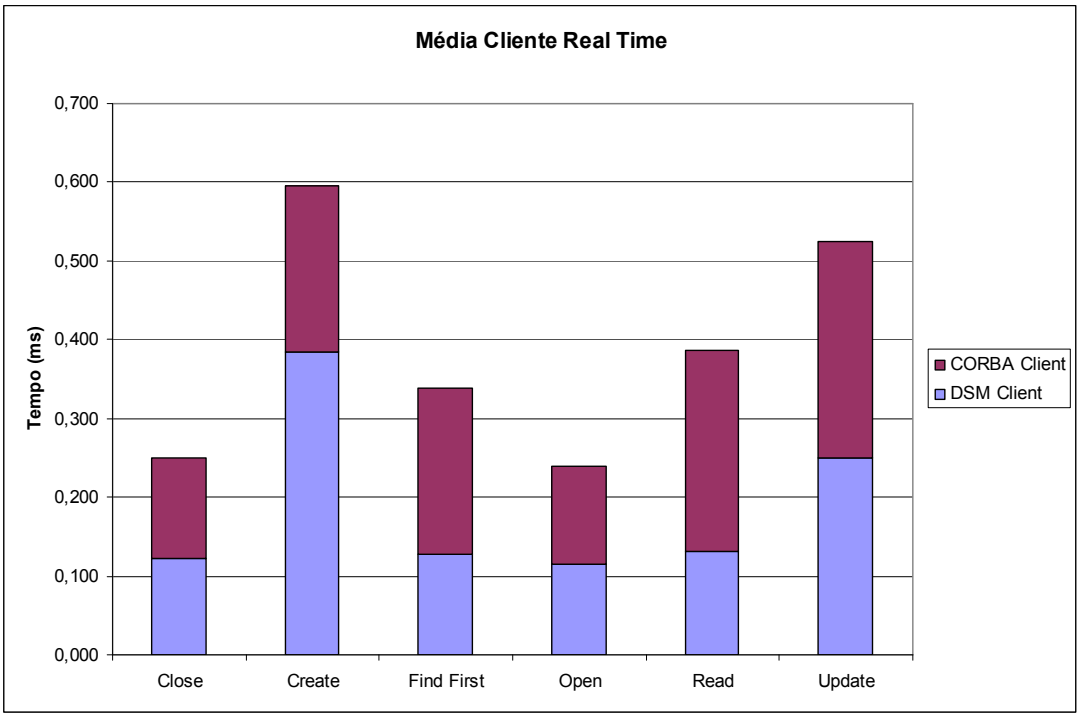

Figura 24 – Média Cliente CORBA Real Time priority class

4.8 Sumário

Neste capítulo foi descrito todo o trabalho desenvolvido no âmbito do projecto, desde a análise de requisitos passando pelo desenho, pela implementação e terminando nos testes. Todas estas fases fazem parte do modelo de desenvolvimento de software em V.

O próximo e último capítulo apresenta as conclusões finais do projecto, mais precisamente as principais contribuições que o projecto teve, como uma perspectiva de trabalho futuro que provenha da realização do mesmo.

Capítulo 5

Conclusão

Este último capítulo do documento pretende apresentar as principais conclusões retiradas da realização do projecto.

Esta apresentação das conclusões divide-se em duas partes, as principais contribuições que a realização do projecto teve, bem como uma perspectiva de trabalho futura que advenha da realização do mesmo.

5.1 Principais Contribuições

O objectivo principal deste projecto era criar uma prova de conceito, de modo a antever alterações necessárias no *Middleware Basic System* ao integrar o CORBA. Esta prova de conceito foi efectuada sobre um dos produtos do *Basic System* o *Data Store Manager*.

Com a realização deste projecto foi possível efectuar interacções com o *Basic System* mais precisamente com as bases de dados por ele controladas. Sendo assim foi possível criar um cliente para o qual a existência do *Basic System* fosse transparente e que mesmo assim conseguisse utilizar as funções pretendidas.

Outra contribuição proveniente do trabalho realizado foi o facto de conseguir, através da utilização do CORBA, a interoperabilidade do *middleware*, deste modo utilizando o CORBA juntamente com o *Basic System* é possível ter aplicações desenvolvidas em várias linguagens a comunicar com o *Basic System*.

No entanto foi verificado um acréscimo de tempo na execução das funções tal como descrito em 4.7.2 Tendo em conta que o *Basic System* está integrado num ambiente em que o sistema tem limites de tempo de resposta curtos, a adição da camada CORBA e o tempo adicional de resposta trazido por esta, pode tornar-se um problema, especialmente se este esta camada for adicionada aos restantes produtos. É necessário

pesar se a interoperabilidade realmente compensa em relação ao aumento do tempo de resposta.

Com todo este trabalho foi provado, tal como era o objectivo inicial do projecto, que é possível dar continuidade a este *legacy system* utilizando uma tecnologia mais recente.

5.2 Perspectiva Futura

A primeira tarefa a realizar após a conclusão do projecto de estágio, é a de identificar qual a causa para aos picos de tempo referidos em 4.7.2 Perceber a causa deste problema não estava no âmbito deste projecto, mas é importante futuramente perceber o porque disto ocorrer e como resolver este problema.

Posteriormente será necessário finalizar a implementação do canal de eventos, de modo a que se consiga efectuar testes de desempenho para este serviço.

Após se ter percebido a causa dos atrasos e terminado a implementação do canal de eventos e visto esta prova de conceito apenas incidir sobre algumas funções de um dos produtos assentes em *Basic System*, pode-se avançar para uma implementação CORBA para os restantes produtos, bem como na implementação de todas as funcionalidades.

Deste modo, e tendo realizado adaptadores CORBA para todos os produtos assentes em *Basic System*, será possível analisar ainda mais precisamente como será o desempenho destas duas camadas juntas.

Durante a concretização dos adaptadores CORBA será necessário efectuar testes bastante exaustivos. Sendo que o *Basic System* é um produto que neste momento se encontra bastante estabilizado, a adição de uma nova camada necessita de continuar a garantir o comportamento correcto da camada já existente.

Outra possível abordagem de modo a evitar a queda de desempenho que a nova camada do *middleware* CORBA traz, é substitui todo o *middleware Basic System* por CORBA. Assim é possível trazer a interoperabilidade desejada melhorando o desempenho. No entanto tendo em conta que estamos num contexto de um sistema crítico esta alteração necessita ser planeada a longo prazo.

Referências

1. Software Process Models. http://www.the-software-experts.de/e_dta-sw-process.htm.

2. Fundamentals of the V Model. http://v-modell.iabg.de/v-modell-xt-html-english/ce15fb4b096b90.html#toc5.

3. Blanken, M. den. *Basic System A standard application platform, services & protocols, basic building blocks (Tutoriais)*. .

4. Bolton, F. *Pure CORBA*. 2002.

5. Bucanac, C. *The V-Model*. 1999.

6. Cardoso, J. *Programaçao em Sistemas Distribuidos*. Lisboa, 2008.

7. Gelbmann, R. Evaluation and Comparison of CORBA Object Request Broker Implementations. 2002, 94.

8. NAV Portugal E.P.E. *Middleware Software Architecture Specification*. 2007.

9. NAV Portugal E.P.E. *LISATM Software Architecture Specification*. 2010.

10. NAV Portugal E.P.E. Website Nav Portugal E.P.E. http://www.nav.pt.

11. NAV Portugal E.P.E. *Prestação de Serviços de Desenvolvimento de Sistemas (Apresentação Power Point)*. .

12. OMG. *The Common Object Request Broker: Architecture and Specification Version 2.3.1*. 1999.

13. OMG. *C++ Language Mapping*. 1999.

14. OMG. *IDL to Java Language Mapping Speciffication*. 1999.

15. OMG. *Data Distribution Service for Real-Time Systems*. 2007.

16. OMG. *OMG Unified Modeling Language - Capitulo 16 (Use Cases) (OMG UML)*. 2011.

17. OMG. OMG's CORBA Website. http://www.corba.org/.

18. Oracle. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html.